



Instituto Politécnico Nacional

**INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
ESCOM**

TRABAJO TERMINAL

"Diseño de un sistema autónomo para monitoreo"

2024 - B125

Presenta:

**HERNÁNDEZ VERGARA EDUARDO
ROJAS CRUZ JOSÉ ÁNGEL**

Directores:

Dr. Juárez Martínez Genaro

Dr. Oliva Morena Luz Noe

MÉXICO CDMX, 11 DE DICIEMBRE DE 2024





Instituto Politécnico Nacional



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
SUBDIRECCIÓN ACADÉMICA

Trabajo Terminal No. 2024 - B125

11 de Diciembre de 2024

DOCUMENTO TÉCNICO

"Diseño de un sistema autónomo para monitoreo"

Trabajo Terminal No. 2024 - B125

PRESENTAN:

Hernández Vergara Eduardo¹
Rojas Cruz José Ángel²

DIRECTORES:

Dr. Juárez Martínez Genaro

Dr. Oliva Morena Luz Noe

Resumen -

Este proyecto propone el diseño e implementación de un autómata capaz de determinar trayectos en espacios euclidianos. La aplicación de este sistema aborda desafíos en navegación autónoma y diseño de redes en tiempo real. Su relevancia se destaca en contextos como en sistemas autónomos. El autómata estará implementado en un modelo bidimensional no lineal.

Este autómata podría servir para el monitoreo de entidades poblacionales y sistemas relacionados.

Palabras clave - Autómata, Control, Diseño de Redes, Ruteos, Tiempo Real.

¹eduardo.hernandezv@proton.me

²angelrojas@gmail.com

Advertencia

"Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan." La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen. Información adicional sobre este reporte técnico podrá obtenerse en: La Subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n Teléfono: 57296000, extensión 52000.

Índice

1. Introducción	6
1.1. Objetivo	6
1.1.1. Objetivo general	6
1.1.2. Objetivos específicos	7
1.2. Justificación	7
2. Marco teórico	8
2.1. Introducción a los autómatas celulares	8
2.1.1. Teoría de autómatas	8
2.1.2. Autómatas celulares de una dimensión	9
2.1.3. Condiciones frontera	10
2.1.4. Vecindario	11
2.1.5. Definición formal	12
2.1.6. Autómatas celulares de 2 dimensiones	13
2.1.7. Ejemplo	15
2.1.8. Entropía de Shannon	16
2.1.9. Sistemas dinámicos	16
2.2. Physarum Polycephalum	18
2.2.1. Mixomiceto	18
2.2.2. Ciclo de vida	18
2.2.3. Physarum Polycephalum	21
2.2.4. El Physarum Polycephalum visto desde la perspectiva computacional	22
2.3. Modo gráfico	23
2.3.1. Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)	24
2.4. Raspberry Pi	25
2.4.1. Historia y evolución	25
2.4.2. Comparativa	27
2.4.3. Raspberry Pi 4 Model B	29
2.5. Protocolos de comunicación	30
2.5.1. WebSocket	30
2.5.2. Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP)	31
2.5.3. Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC)	32
3. Estado del Arte	32
3.1. Physarum Polycephalum	32
3.1.1. Modelado de Adamatzky	33
3.1.2. Modelado Guillermo Olvera	34
3.1.3. Modelado de Yair Marin	35
3.1.4. Modelado de Jeff Jones	37
3.1.5. Modelado de Gunji	38
3.2. Robots para monitoreo poblacional	39
3.2.1. Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis	39
3.2.2. Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields	40

3.2.3.	The Role of Robots in Environmental Monitoring	40
4.	Propuesta a desarrollar	42
4.1.	Requerimientos	42
4.1.1.	Requerimientos funcionales	42
4.1.2.	Requerimientos no funcionales	43
4.2.	Diagramas	44
5.	Implementación	53
5.1.	Simulador del Physarum Polycephalum	53
5.1.1.	Generación de rutas de nuestro simulador	59
5.1.2.	Codificación e implementación de algoritmo en el robot en la primera iteración 1	66
5.1.3.	Codificación e implementación de algoritmo en el robot en la primera iteración 2	67
5.1.4.	Ajustes del algoritmo basados en pruebas unitarias 1	67
5.1.5.	Ajustes del algoritmo basados en pruebas unitarias 2	68
5.1.6.	Diseño inicial de interfaz para control y monitorear	68
5.1.7.	Ajustes de rutas basado en resultado de pruebas de aceptación 1	69
5.2.	Robot Propuesto	71
5.2.1.	Desarrollo inicial del sistema del control del robot	72
5.2.2.	Ajustes de código en función de pruebas unitarias y de aceptación 1	80
5.2.3.	Ajustes de código en función de pruebas unitarias y de aceptación 2	89
5.2.4.	Ajustes de código en función de las pruebas de carga 1	98
5.2.5.	Ajustes de código en función de pruebas unitarias y de aceptación 3	109
5.2.6.	Ajustes de código en función de las pruebas de carga 2	126
6.	Pruebas del sistema	145
6.1.	Pruebas unitarias del algoritmo en simulación de mapas 1	146
6.2.	Evaluación de desempeño del software en pruebas iniciales 1	146
6.3.	Recopilación de datos del software y desempeño 1	152
6.4.	Redacción del informe técnico inicial basado en la Iteración 1	157
6.5.	Pruebas unitarias del algoritmo en simulación de mapas 2	157
6.6.	Pruebas de aceptación en escenarios pequeños	158
6.6.1.	1ra Prueba de aceptación: Elección de estados a través del teclado.	158
6.6.2.	2da Prueba de aceptación: Colocación de los estados inicial y final.	158
6.6.3.	3ra Prueba de aceptación: Inicialización de la simulación.	159
6.7.	Evaluación de desempeño del software en pruebas iniciales 2	159
6.8.	Recopilación de datos del software y desempeño 2	162
6.9.	Pruebas de aceptación en entornos medianos	171
6.9.1.	1ra Prueba de aceptación: Modificación del tamaño del lienzo	171
6.9.2.	2da Prueba de aceptación: Elección de estados a través del teclado.	171
6.9.3.	3ra Prueba de aceptación: Colocación de los estados inicial y final.	172
6.9.4.	4ta Prueba de aceptación: Inicialización de la simulación.	172
6.10.	Pruebas de aceptación en entornos complejos simulados 1	173
6.10.1.	1ra Prueba de aceptación: Modificación del tamaño del lienzo	173
6.10.2.	2da Prueba de aceptación: Carga de mapas	173
6.10.3.	3ra Prueba de aceptación: Elección de estados a través del teclado.	174
6.10.4.	4ta Prueba de aceptación: Colocación de los estados inicial y final.	174
6.10.5.	5ta Prueba de aceptación: Inicio de la simulación	175
6.11.	Evaluación final del desempeño del software 1	175

6.12. Recopilación y análisis de datos del rendimiento del robot de IEEE 1872.1 e IEEE 2914	179
6.13. Redacción del informe con mejoras de la Iteración 2	180
6.14. Recopilación final de datos del software para el informe 1	181
6.15. Documentación del avance segunda iteración	181
6.16. Pruebas de aceptación en entornos complejos simulados 2	182
6.17. Evaluación final del desempeño del software 2	182
6.18. Recopilación final de datos del software para el informe 2	182
6.19. Informe final con resultados de la Iteración 3	185
6.20. Pruebas de aceptación en entornos complejos simulados 3	186
6.20.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo	186
6.20.2. 2da Prueba de aceptación: Carga de mapas	186
6.20.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.	187
6.20.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.	187
6.20.5. 5ta Prueba de aceptación: Inicio de la simulación	188
6.21. Evaluación final de desempeño del software 3	188
6.22. Recopilación final de datos del software para el informe 3	189
6.23. Análisis de primera iteración de datos de sensores	191
6.24. Diseño de módulos de hardware del robot para pruebas de integración	193
6.24.1. Módulo de actuadores (Motores)	193
6.24.2. Módulo de sensores (LiDAR)	194
6.24.3. Unidad de control (Raspberry Pi 4 B)	194
6.24.4. Módulo de visualización	195
6.24.5. Pruebas de integración	195
6.25. Desarrollo de módulos de hardware del robot para pruebas de integración	197
6.25.1. Módulo de actuadores	197
6.25.2. Módulo de sensores	198
6.25.3. Unidad de control	199
6.25.4. Sistema de visualización	208
6.25.5. Integración de módulos	208
6.26. Evaluación de primera iteración del desempeño del robot de pruebas integrales	212
6.27. Pruebas unitarias e integración de sensores y motores en entornos controlados	212
6.27.1. Pruebas unitarias	212
6.27.2. Pruebas de integración	217
6.28. Pruebas de aceptación en escenarios controlados con retroalimentación de los sensores	219
6.29. Análisis de segunda iteración de datos de sensores para implementación de interfaz gráfica	220
6.30. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos	221
6.31. Pruebas de aceptación en entornos complejos simulados y reales	222
6.32. Evaluación de segunda iteración del desempeño del robot de pruebas integrales	223
6.33. Análisis tercera iteración de datos de sensores para implementación de aplicación móvil	223
6.34. Pruebas de aceptación en entornos complejos simulados y reales	225
6.35. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos	225
6.36. Evaluación de tercera iteración del desempeño del robot de pruebas integrales	226
6.37. Redacción del informe final	226

7. Trabajo a futuro y conclusiones

228

Abstract - This project proposes the design and implementation of an automaton capable of determining trajectories in Euclidean spaces. The application of this system addresses challenges in autonomous navigation and real-time network design. Its relevance is highlighted in contexts such as in autonomous systems. The automaton will be implemented in a two-dimensional nonlinear model. This automaton could serve for the monitoring of population entities and related systems.

1. Introducción

En el complejo mundo actual, en el cual los procesos industriales se valen de la automatización y digitalización, los autómatas celulares emergen como sistemas dinámicos discretos de gran potencial. Dichos sistemas están constituidos por matrices de celdas; estas celdas son la unidad básica de los autómatas celulares y cada celda puede estar en un estado determinado, como 1 o 0, vivo o muerto. En realidad, pueden tener múltiples interpretaciones, pero básicamente es un sistema binario. Además, estas matrices de celdas se rigen por las reglas del autómatas, que evolucionan conforme lo hacen las generaciones. En este proyecto nos enfocaremos en el desarrollo y aplicación de un autómatas programable en una Raspberry Pi, con el objetivo de proporcionar a las organizaciones una solución integral y adaptable. Esta solución les permitirá el monitoreo y control eficiente de sistemas y procesos en tiempo real.

La necesidad de un monitoreo en tiempo real es muy alta en el entorno social, empresarial y en algunas aplicaciones para uso gubernamental, donde la agilidad y la eficacia son esenciales. En esta situación, los autómatas celulares se presentan como herramientas bastante versátiles a la hora de interactuar dinámicamente con su entorno. En términos simples, cada uno de los elementos en un autómatas celular se relaciona con sus vecinos (celdas que se encuentran alrededor de nuestra celda actual), y su estado en la próxima generación se determina según el estado de sus vecinos en la generación actual. Esta capacidad única de interacción y cambio dinámico de estados los convierte en instrumentos poderosos para abordar una variedad de desafíos.

Este proyecto no solo se enfoca en la implementación técnica de un autómatas celular, sino también en su aplicación práctica en entornos específicos. Los autómatas celulares han demostrado su valía en diversas áreas, desde el monitoreo y predicción del cambio de uso de la tierra hasta la planificación de rutas sin colisión para robots. Dos referencias particulares, [1] y [2], destacan por su relevancia directa a nuestra propuesta de Trabajo Terminal (TT), ya que se centran en tareas de monitoreo y la implementación de robots sin colisiones, respectivamente. Justamente en nuestro caso, es el monitoreo y la prevención de colisiones de robots.

En este enfoque buscamos resaltar la versatilidad de los autómatas celulares como herramientas de solución aplicables en situaciones del mundo real. A su vez, se espera que el autómatas sea adaptable para que pueda ser implementado en sectores emergentes, como la Inteligencia Artificial (IA), ampliando aún más su alcance y utilidad.

1.1. Objetivo

1.1.1. Objetivo general

Implementar un autómatas que sea capaz de determinar sus trayectos en espacios bidimensionales para monitorear trazando rutas en tiempo real.

1.1.2. Objetivos específicos

- Diseñar un autómata basado en el mixomiceto *Physarum polycephalum* que sea capaz de determinar trayectos en espacios bidimensionales.
- Implementar una simulación del autómata en un programa desarrollado en el lenguaje de programación C++.
- Implementar el autómata en robot cuyo controlador sea una Raspberry Pi 4.
- Diseñar un sistema de monitoreo que permita visualizar el estado del autómata y del robot.
- Realizar las pruebas en un entorno controlado.
- Realizar las pruebas en un entorno real.

1.2. Justificación

En el marco actual de la automatización constante de procesos y el desarrollo de herramientas que facilitan la realización de tareas, han surgido distintas tecnologías y procesos que han sabido atacar de la mejor manera las problemáticas que involucran a la automatización. Sin embargo, muchas veces es complicado darles un correcto seguimiento a las actividades y procesos que están involucrados en la realización de tareas automáticas, ocasionando distintos problemas que afectan en la solución del problema al que originalmente planteaban solucionar o complicar de manera innecesaria el proceso. Por lo que es necesario crear herramientas de monitorización con métodos más fiables a los ya existentes, y que, además de brindar un correcto seguimiento a los procesos a los cuales se les hace un análisis, tenga la capacidad de reaccionar ante los cambios relevantes e importantes que surjan durante la realización de las distintas tareas y procesos en los que se vea involucrado.

Es por eso por lo que se busca la implementación de un sistema robótico, el cual, por medio de la aplicación de la teoría de autómatas celulares, monitorice la realización de distintas tareas y procesos en los que se vea involucrado y, además, priorizando siempre que la monitorización sea efectiva, continua y confiable. Esto para las distintas industrias que realicen actividades en las cuales se vean involucradas la automatización de procesos y tareas.

Los autómatas celulares han sido usados para distintas disciplinas que van desde la antropología hasta los gráficos por computadora, sin embargo, ha sido muy poco visto en actividades que involucren sistemas robóticos debido al comportamiento y la manera en la que los autómatas celulares se comportan a partir de diversas entradas, siendo a veces complicado discernir el comportamiento que se tendrá, lo que añade complejidad al implementar uno de estos autómatas a sistemas robóticos. Pero gracias al conocimiento brindado por algunas materias como lo son Sistemas Operativos, Arquitectura de Computadoras, Diseño Digital y Teoría Computacional, se puede llegar a un procedimiento y tratamiento de la información tal que sea posible dirigir el autómata a la mejor solución posible.

2. Marco teórico

Para avanzar en nuestro proyecto, conforme a los **Objetivos específicos** previamente definidos, es esencial comprender los fundamentos de los *autómatas celulares*, incluyendo sus características y propiedades. Esta comprensión es clave para su implementación en nuestro proyecto. Por ello, dedicaremos esta sección a detallar estos conceptos básicos, características y propiedades de los autómatas celulares. Asimismo, proporcionaremos una introducción al mixomiceto *Physarum polycephalum*, destacando su conexión con los autómatas celulares.

Adicionalmente, subrayaremos el papel crucial de la *Raspberry Pi 4*, que se encarga de gestionar el robot y de aplicar el autómata celular. Incluirá también una descripción concisa de la librería gráfica *Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)* (o *Vulkan*), seleccionada para la simulación del autómata celular.

2.1. Introducción a los autómatas celulares

Primero es necesario conocer la teoría de autómatas y como se relaciona con los autómatas celulares, por ello daremos un breve repaso de la teoría de autómatas.

2.1.1. Teoría de autómatas

La teoría de autómatas es el estudio de dispositivos de cálculo abstractos, es decir de las máquinas.[3] Estos autómatas son modelos matemáticos fundamentales en el área de estudio de las ciencias de la computación, son usados para entender los procesos de cálculo y toma de decisiones. En la teoría de autómatas se estudian los autómatas finitos, los autómatas con pila, las máquinas de Turing, los autómatas celulares, etc. Los autómatas regulares pueden ser jerarquizados en una jerarquía de Chomsky, que es una jerarquía de lenguajes formales. La cual sería la siguiente[4]:

- **Tipo 3 - Gramáticas regulares:** Estos generan los lenguajes regulares. Estas se restringen a producciones de la forma $A \rightarrow a\gamma$ y $A \rightarrow aB$. Son asociados a los autómatas finitos.
- **Tipo 2 - Gramáticas libres de contexto:** Estos generan los lenguajes independientes del contexto. Estas se restringen a producciones de la forma $A \rightarrow \gamma$. Son asociados a los autómatas con pila.
- **Tipo 1 - Gramáticas sensibles al contexto:** Estos generan los lenguajes sensibles al contexto. Estas se restringen a producciones de la forma $\alpha A\beta \rightarrow \alpha\gamma\beta$. Son asociados a las máquinas de Turing linealmente acotadas (significa que la cinta de la máquina de Turing tiene un límite determinada por un cierto número entero de veces sobre la longitud de entrada).
- **Tipo 0 - Gramáticas irrestrictas:** Estos generan los lenguajes recursivamente enumerables. Estas se restringen a producciones de la forma $\alpha A\beta \rightarrow \delta$. Son asociados a las máquinas de Turing.

En cambio los autómatas celulares, aunque diferentes en estructura y aplicación a las gramáticas formales, también son modelos matemáticos fundamentales en el área de estudio de las ciencias de la computación y forman parte de la teoría de autómatas. Mientras que los autómatas convencionales se centran en el procesamiento secuencial de cadenas de símbolos y operan basándose en estados y transiciones claramente definidos, los autómatas celulares utilizan una

red de células cuyos estados evolucionan en paralelo, siguiendo reglas locales. Esta diferencia fundamental en su enfoque los hace especialmente adecuados para modelar y explorar fenómenos que involucran procesos dinámicos y patrones espaciales. A pesar de estas diferencias, los autómatas celulares se alinean con los principios fundamentales de la teoría de autómatas en cuanto a la representación y manipulación de información, ofreciendo una perspectiva más amplia y diversa sobre lo que constituye un ‘*autómata*’ en el contexto de la computación y el procesamiento de información. Su inclusión en la teoría de autómatas subraya la amplitud y la profundidad de este campo, demostrando que la teoría de autómatas no solo se limita a las máquinas y lenguajes formales tradicionales, sino que también abarca modelos computacionales más generales y versátiles.

Una vez que hemos explicado la teoría de automatas podemos pasar a explicar en mas detalle los autómatas celulares.

2.1.2. Autómatas celulares de una dimensión

Los autómatas celulares de una dimensión consisten de una línea de celdas o estados, cada una con 2 estados posibles, 0 o 1, vivo o muerto, etc. Estas celdas se actualizan en cada generación, de acuerdo a una regla de evolución, la cual determina el estado de una celda en la siguiente generación, basándose en el estado de la celda y sus vecinos en la generación actual. La regla de evolución se aplica a todas las celdas de la misma manera, y en paralelo, es decir, todas las celdas se actualizan al mismo tiempo. Para ejemplificar esto, se puede ver la Figura 1. En donde tenemos la regla de evolución 30, de las *Reglas de Wolfram*[5], o *Automatas Celulares Elementales* (Elemental Cellular Automata, ECA), en la cual se puede ver que la celda de la generación $t + 1$ depende de la celda de la generación t y sus vecinos.

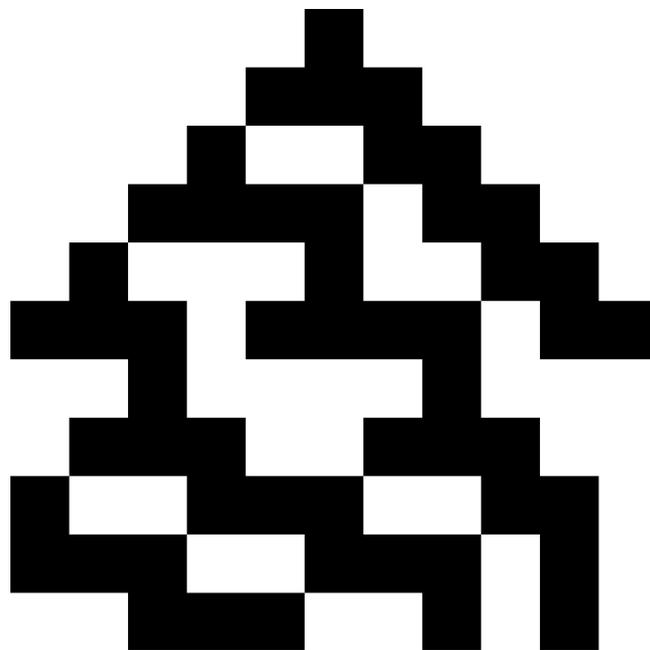


Figura 1: Regla de evolución 30 de las *Reglas de Wolfram* 10 generaciones o (t)

Como podemos ver en estos autómatas celulares de una dimensión los que podemos considerar los elementos más importantes son:

- **Vecindad:** Es el conjunto de celdas que se toman en cuenta para la actualización de una celda.

- **Estado:** Es el valor que puede tomar una celda, en este caso solo puede ser 0 o 1.
- **Regla de evolución:** Es la regla que determina el estado de una celda en la siguiente generación.

En este caso la vecindad la forman la celda y sus dos vecinos, pero puede ser de cualquier tamaño, siempre y cuando sea simétrica, es decir, que la celda se encuentre en el centro de la vecindad. El estado de la celda puede ser cualquier valor, pero en este caso solo puede ser 0 o 1. La regla de evolución es la que determina el estado de la celda en la siguiente generación, en este caso la regla de evolución 30, que se puede ver en la Figura 1, es la siguiente:

111	110	101	100	011	010	001	000
0	0	0	1	1	1	1	0

Cuadro 1: Regla de evolución 30

Como podemos ver en el Cuadro 1 la regla de evolución 30 es una regla de evolución local, es decir, que solo depende de la celda y sus vecinos, y no de toda la línea de celdas. En este caso la regla de evolución 30 es una regla de evolución determinista, es decir, que para una celda y sus vecinos siempre se obtiene el mismo resultado. Pero también existen las reglas de evolución no deterministas, en las cuales para una celda y sus vecinos se puede obtener más de un resultado. En este caso la regla de evolución 30 es una regla de evolución unidimensional, es decir, que la celda solo depende de sus vecinos de la izquierda y de la derecha, pero también existen las reglas de evolución bidimensionales, n-dimensionales, etc.

2.1.3. Condiciones frontera

Por definición los autómatas celulares son infinitos, pero en la práctica no se pueden tener autómatas celulares infinitos, por lo que se tienen que definir condiciones frontera, las cuales son las condiciones que se tienen en los extremos del autómata celular. Existen diferentes tipos de condiciones frontera, las cuales son:

- **Abierta** En este caso las celdas de los extremos no tienen vecinos, por lo que no se pueden actualizar.
- **Periódica** En este caso las celdas de los extremos tienen como vecinos a las celdas del otro extremo.
- **Reflejante** En este caso las celdas de los extremos tienen como vecinos a las celdas del otro extremo, pero invertidas.
- **Frontera** En este caso las celdas de los extremos tienen como vecinos a celdas con un valor fijo.

En nuestro caso, utilizaremos la condición frontera periódica; esta tiene forma de un toroide, como se puede ver en la Figura 2.

También podemos observar que en la Figura 2 se puede ver que las celdas de los extremos tienen como vecinos a las celdas del otro extremo, por lo que se puede decir que es una condición frontera periódica.

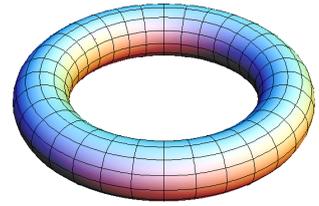


Figura 2: Representación de un toroide [6]

2.1.4. Vecindario

Un vecindario es un conjunto de celdas que se toman en cuenta para la actualización de una celda. Como vimos con anterioridad en los autómatas celulares de una dimensión, el vecindario de una celda es la celda y sus dos vecinos, pero en los autómatas celulares de dos dimensiones el vecindario de una celda puede ser de cualquier tamaño, siempre y cuando sea simétrico, es decir, que la celda se encuentre en el centro del vecindario. En la Figura 3 se puede ver un ejemplo de un vecindario de una celda. E incluso en los autómatas celulares de dos dimensiones se pueden tener vecindarios de diferentes formas, como se es común verlos en forma cuadrada y en forma hexagonal, así como se muestra en las Figuras 3 - 7.

A su vez los vecindarios más usados son los siguientes:

- **Vecindario de Moore** En este caso el vecindario de una celda es la celda y sus ocho vecinos.

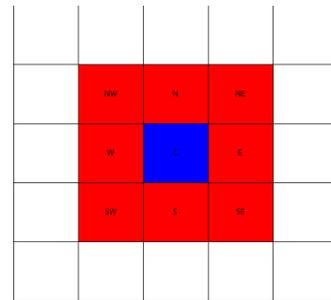


Figura 3: Vecindario de Moore

- **Vecindario de von Neumann** En este caso el vecindario de una celda es la celda y sus cuatro vecinos.

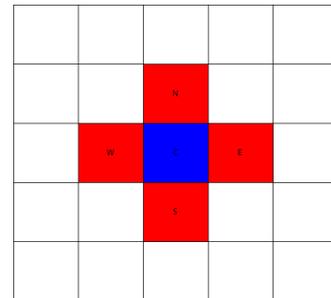


Figura 4: Vecindario de von Neumann

- **Vecindario de Moore extendido** En este caso el vecindario de una celda es la celda y sus veinticuatro vecinos.

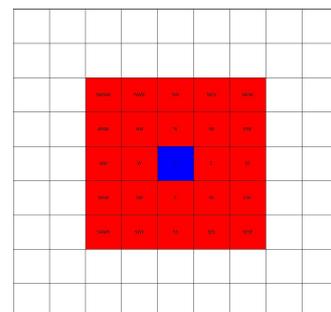


Figura 5: Vecindario de Moore extendido

Vecindario de von Neumann extendido En

- este caso el vecindario de una celda es la celda y sus doce vecinos.

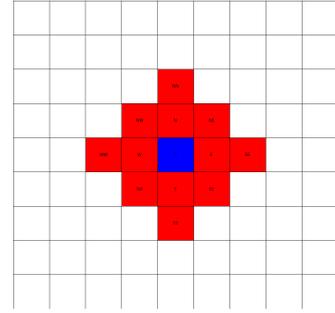


Figura 6: Vecindario de von Neumann extendido

Vecindario de hexagonal En este caso el ve-

- cindario de una celda es la celda y sus seis vecinos.

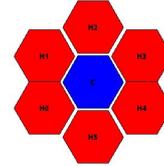


Figura 7: Vecindario de hexagonal

Una vez que hemos explicado eso podemos pasar a definir formalmente los autómatas celulares.

2.1.5. Definición formal

Primero denotemos \mathbb{Z} como el conjunto de los números enteros, es decir, $\mathbb{Z} = (-\infty, -1, 0, 1, \infty)$. y la longitud de cualquier tupla x como $|x|$. Para todas las tuplas x y y de la misma longitud, denotemos $x \oplus y$ como la tupla que resulta de la suma componente a componente de x y y , es decir, $(x \oplus y)_i = x_i + y_i$ para todo $i \in \mathbb{Z}$.

Entonces tenemos que un autómata celular es una tupla (\mathbb{Z}^n, S, N, f) tal que la n dimensión es al menos 1 donde $n \in \mathbb{Z}^+$, S es un conjunto finito no vacío de estados, N es un conjunto finito no vacío de vecindades perteneciente a \mathbb{Z}^n y f es una función de transición local, es decir, $f : S^N \rightarrow S$ donde S^N representa al conjunto de todas las posibles configuraciones de vecindad en N .

La configuración inicial de un autómata celular es una función $c : \mathbb{Z}^n \rightarrow S$ que asigna un estado a cada celda. La evolución de un autómata celular es una función $F : S^{\mathbb{Z}^n} \rightarrow S^{\mathbb{Z}^n}$ que asigna una configuración a la siguiente configuración, es decir, $F(c) = c'$, donde c' es la configuración resultante de aplicar la función de transición local a cada celda de la configuración c , es decir, $c'(x) = f(c|_{x+N})$ para todo $x \in \mathbb{Z}^n$, donde $c|_{x+N}$ es la restricción de c a la vecindad $x + N$. Esto también aplica n dimensionalmente, es decir, que se podría decir que $c'(x, y) = f(c|_{(x,y)+N})$ para todo $(x, y) \in \mathbb{Z}^2$. Otra notación que podemos usar, y de hecho es la que utilizaremos es $C(x, y : t)$ donde C es el centro de la vecindad, x, y son las coordenadas de la celda y t es el tiempo, o generaciones.

Cabe añadir que puede haber restricciones adicionales en el conjunto de vecindades N y en la función de transición local f . Por ejemplo, en el caso de los autómatas celulares de una dimensión, el conjunto de vecindades N es un conjunto de tuplas de longitud 3, donde la primera componente es la celda, la segunda componente es la celda de la izquierda y la tercera

componente es la celda de la derecha. Y la función de transición local f es una función de 8 variables booleanas, es decir, $f : \{0, 1\}^3 \rightarrow \{0, 1\}$.

Y en el caso de los autómatas celulares de dos dimensiones, el conjunto de vecindades N es un conjunto de tuplas de longitud variable, dependiendo del tipo de vecindad, donde la primera componente es la celda y las demás componentes son las celdas vecinas. Por ejemplo, en el caso de la vecindad de Moore, el conjunto de vecindades N es un conjunto de tuplas de longitud 9, donde la primera componente es la celda (x, y) el cual sería el centro de la vecindad, la segunda componente es la celda $(x - 1, y - 1)$, la tercera componente es la celda $(x - 1, y)$, la cuarta componente es la celda $(x - 1, y + 1)$, la quinta componente es la celda $(x, y - 1)$, la sexta componente es la celda $(x, y + 1)$, la séptima componente es la celda $(x + 1, y - 1)$, la octava componente es la celda $(x + 1, y)$ y la novena componente es la celda $(x + 1, y + 1)$. Aquí (x, y) es la celda central de la vecindad. Y la función de transición local f es una función de 512 variables booleanas, es decir, $f : \{0, 1\}^9 \rightarrow \{0, 1\}$.

Esta definición formal de autómata celular fue tomada de [7]. Una vez explicada la definición formal de autómata celular, podemos pasar a explicar a más detalle los autómatas celulares de 2 dimensiones, los cuales son los que se usan en este Trabajo Terminal (TT).

2.1.6. Autómatas celulares de 2 dimensiones

Los autómatas celulares de 2 dimensiones son los que se usan en este Trabajo Terminal (TT), por ello es necesario explicarlos con más detalle. Primero recordando lo ya explicado en la sección 2.1.5, un autómata celular de 2 dimensiones es una tupla (\mathbb{Z}^2, S, N, f) y este necesita de 8 elementos para poder ser definido, los cuales son los siguientes:

1. **Celdas:** Es la unidad básica del autómata celular. Cada celda ocupa una posición en el espacio, para ser representados suelen usarse cuadrículas o redes, esta tiene un estado y una vecindad.
2. **Estados:** Cada celda puede estar en uno de varios estados posibles. En los autómatas celulares más simples, cada celda puede estar en uno de dos estados posibles (0 o 1, vivo o muerto, etc), pero en los autómatas celulares más complejos, cada celda puede estar en uno de varios estados posibles. En el caso en particular del Physarum polycephalum, cada celda puede estar en uno de nueve estados posibles $\mathbb{P} = \{x \in \mathbb{Z} \mid 0 \leq x \leq 8\}$ entonces es $S = \mathbb{P}$.
3. **Cuadrícula o Red:** Las celdas están dispuestas a lo largo del espacio euclidiano, estas suelen disponerse en una cuadrícula o red, en donde cada celda ocupa una posición en el espacio. Es n-dimensional, pero en este caso es 2-dimensional, es decir, $n = 2$.
4. **Vecindad:** Es el conjunto de celdas que se toman en cuenta para la actualización de una celda. En el caso de la vecindad de Moore, que se puede ver en la sección 2.1.4, esta es $N = 8$
5. **Reglas de Transición:** Son un conjunto de reglas que determinan como cambia el estado de la célula en función del estado actual de ella y de sus vecinos. Estas reglas se aplican repetidamente a lo largo del tiempo, generalmente de manera síncrona, es decir, todas las celdas se actualizan al mismo tiempo. Estas están definidas por la función de transición local f . Ejemplificando lo anterior tenemos que en el juego de la vida de Conway [8] donde tenemos que $C(x, y : t)$ que es la celda central y $N(x, y : t)$ que es la vecindad de Moore

de la celda central, además tenemos que tiene $f : \{0, 1\}^9 \rightarrow \{0, 1\}$, entonces podemos deducir que la función de transición se define como:

$$f(C(x, y : t), N(x, y : t)) = \begin{cases} 1 & \text{si } C(x, y : t) = 0 \text{ y } N(x, y : t) = 3 \\ 1 & \text{si } C(x, y : t) = 1 \text{ y } N(x, y : t) = 2 \text{ o } N(x, y : t) = 3 \\ 0 & \text{en otro caso} \end{cases}$$

6. **Tiempo o Generaciones:** Es el número de veces que se aplica la función de transición local f . En este caso es $t \in \mathbb{Z}^+$.
7. **Condiciones Iniciales:** Antes de que el autómata celular comience a evolucionar, se debe especificar el estado de cada celda. En este caso es $c : \mathbb{Z}^2 \rightarrow S$. Estas condiciones iniciales pueden ser aleatorias o no.
8. **Condiciones Frontera:** Son las condiciones frontera previamente mencionadas en la sección 2.1.2.

2.1.7. Ejemplo

Una vez que hemos explicado los autómatas celulares de 2 dimensiones podemos pasar a explicar un ejemplo de un autómata celular de 2 dimensiones.

En este caso mostraremos un autómata celular de 2 dimensiones (binario) con vecindad de Moore y con una configuración totalística $B4678/S35678$, es decir, una celda nacerá si tiene 4, 6, 7 u 8 vecinos vivos y sobrevivirá si tiene 3, 5, 6, 7 u 8 vecinos vivos. Este autómata celular también es conocido con el nombre de *Anneal* y es mencionado en [9].

Para formalizar tenemos que este autómata $\mathbb{A} = (\mathbb{Z}^2, S, N, f)$ donde: $S = \{0, 1\}$, $N = \{0, 1\}^9$ y $f : \{0, 1\}^9 \rightarrow \{0, 1\}$ y C es la celda central, entonces podemos deducir que la función de transición se define como:

$$f(C(x, y : t), N(x, y : t)) = \begin{cases} 1 & \text{si } C(x, y : t) = 0 \text{ y } N(x, y : t) \in \{4, 6, 7, 8\} \\ 1 & \text{si } C(x, y : t) = 1 \text{ y } N(x, y : t) \in \{3, 5, 6, 7, 8\} \\ 0 & \text{en otro caso} \end{cases}$$

Dándonos como resultado la siguiente evolución en 250 generaciones (véase la Figura 8).

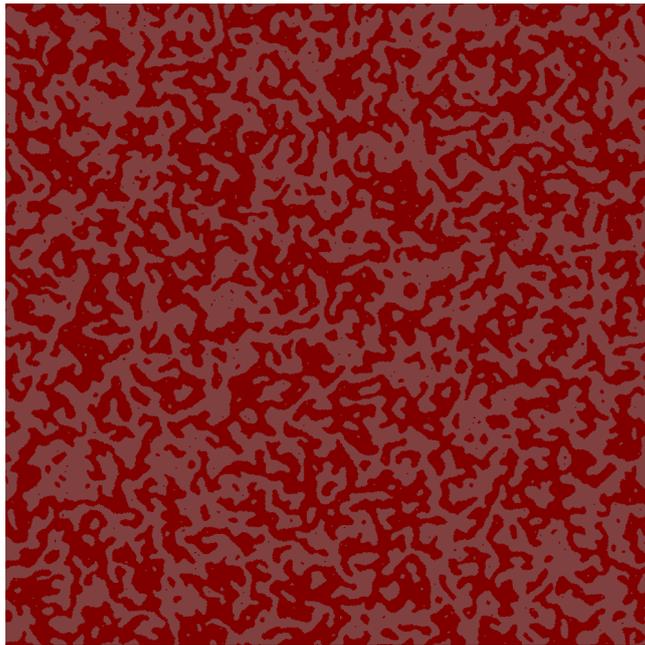


Figura 8: Evolución del autómata celular *Anneal*

2.1.8. Entropía de Shannon

La entropía de Shannon es un concepto fundamental en la teoría de la información, que se utiliza para medir la incertidumbre en una variable aleatoria. La entropía en si es el grado de información/desinformación que tenemos en un sistema. Es decir que cuanto más información tengamos de un sistema, menor será la entropía y viceversa. La entropía de Shannon, nombrada así por Claude Shannon[10], se define matemáticamente la suma negativa de las probabilidades de cada posible valor de la variable aleatoria multiplicada por el logaritmo de la probabilidad de ese valor. Esta definición es la siguiente:

$$H(X) = - \sum_{i=1}^n p(x_i) \log_b p(x_i) \quad (1)$$

Donde X es una variable aleatoria discreta, $p(x_i)$ es la probabilidad de que la variable aleatoria X tome el valor x_i y b es la base del logaritmo. En el caso de que la base del logaritmo sea 2, la unidad de medida de la entropía serán los bits. En el caso de que la base del logaritmo sea e , la unidad de medida de la entropía serán los nat. Y en el caso de que la base del logaritmo sea 10, la unidad de medida de la entropía serán los dits.

En nuestro caso en particular usamos bits como unidad de medida de la entropía, ya que lo usamos para saber que tipo de comportamiento tiene el autómata celular. Es decir, si la entropía es alta, entonces el autómata celular tiene un comportamiento caótico, y si la entropía es baja, entonces el autómata celular tiene un comportamiento ordenado.

Como vimos en ejemplo anterior, el autómata celular *Anneal* podemos observar rápidamente su entropía en la Figura 9:

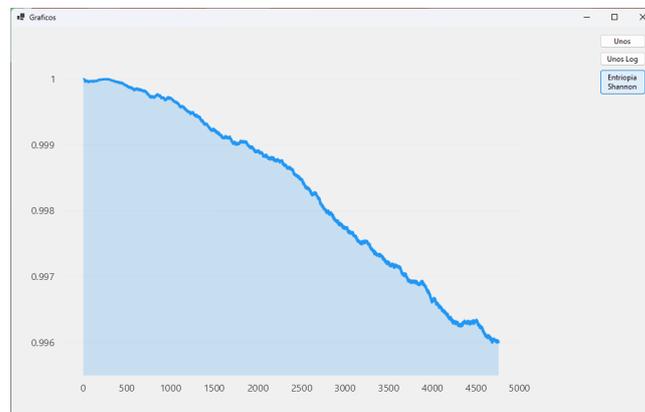


Figura 9: Entropía del autómata celular *Anneal* en 5000 generaciones

2.1.9. Sistemas dinámicos

La teoría de Sistemas dinámicos se puede concebir como un enfoque para describir cómo un estado dado se modifica o progresa hacia otro a medida que transcurre el tiempo, es decir, cómo un sistema evoluciona temporalmente. La teoría de Sistemas dinámicos se puede aplicar a cualquier área de la ciencia, como por ejemplo, la biología, la economía, la física, la química, etc. En nuestro caso en particular, la teoría de Sistemas dinámicos se aplica a los autómatas celulares, ya que los autómatas celulares son sistemas dinámicos discretos.

Nos referimos a sistemas dinámicos discretos cuando el tiempo es discreto. Con los autómatas celulares podemos ver que el tiempo es discreto, ya que el tiempo se mide en generaciones. Es decir, en cada generación el autómata celular evoluciona de un estado a otro. En la Figura 10 podemos ver un ejemplo de la evolución de un autómata celular en el tiempo:

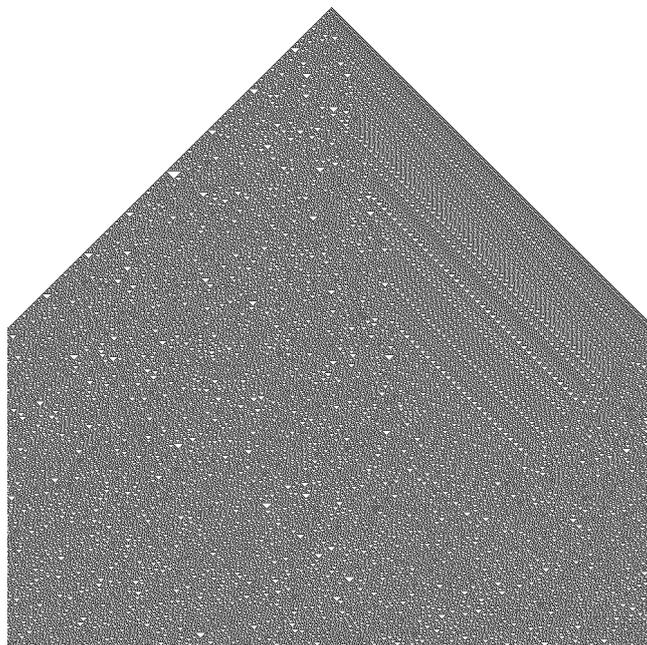


Figura 10: Evolución de la regla 30 en 1000 generaciones o pasos de tiempo

No daremos una explicación más en profundidad de los sistemas dinámicos, ya que no es el objetivo de este trabajo. Para más información sobre sistemas dinámicos y caos, se recomienda leer [11] y también [12]. Sin embargo, lo que sí es importante mencionar es que los sistemas dinámicos pueden ser clasificados, en este caso nos enfocaremos en los sistemas triviales, los sistemas complejos y los sistemas caóticos.

Esto lo hacemos con el propósito de poder clasificar el comportamiento de nuestro autómata celular (Physarum Polycephalum) en base a su comportamiento. Es decir, si el autómata celular se comporta de manera trivial, compleja o caótica. Para esto, primero daremos una breve explicación de cada uno de estos tipos de sistemas dinámicos. A su vez, quisiéramos destacar que para hacer nuestra clasificación o determinar el tipo de comportamiento tiene nuestro autómata celular usaremos una gráfica en la cual podamos observar la Entropía de Shannon en función del tiempo. Ya que para poder usar como método de clasificación los atractores, requerimos que la cantidad de estados en el Physarum Polycephalum es: $\mathbb{P} = \{x \in \mathbb{Z} | 0 \leq x \leq 8\}$ entonces es $S = \mathbb{P}$ y por lo tanto su función de transición es: $f : \mathbb{P} \rightarrow \mathbb{P}$ o sea $f : \mathbb{P}^N \rightarrow \mathbb{P}$ donde N es la vecindad del autómata celular. Por lo tanto, la función de transición es una función de \mathbb{P}^9 , dandonos como resultado un total de 9^9 o sea 387,420,489 estados posibles. Por lo tanto, no es posible graficar todos los estados posibles del autómata celular.

Ahora si, daremos una breve explicación de los sistemas mencionados anteriormente:

- **Sistemas Triviales:** Son sistemas que no tienen comportamiento complejo, es decir, son sistemas que tienen un comportamiento predecible. Por ejemplo, un péndulo simple, ya que su movimiento es periódico y predecible.
- **Sistemas Complejos:** Son sistemas que tienen un comportamiento complejo, es decir, son sistemas que tienen un comportamiento impredecible debido a su sensibilidad a las

condiciones iniciales y a la presencia de interacciones no lineales. Por ejemplo, el clima, ya que es imposible predecir el clima con exactitud.

- **Sistemas Caóticos:** Los sistemas caóticos son un subconjunto de sistemas complejos que son extremadamente sensibles a las condiciones iniciales. Pequeñas variaciones en las condiciones iniciales pueden llevar a resultados completamente diferentes en el tiempo. Los sistemas caóticos pueden parecer aleatorios y desordenados, pero en realidad están gobernados por ecuaciones matemáticas deterministas. Un ejemplo clásico de sistema caótico es el sistema de doble péndulo, donde el movimiento se vuelve impredecible y altamente sensible a las condiciones iniciales después de un corto período de tiempo.

2.2. Physarum Polycephalum

Para poder desarrollar este Trabajo Terminal (TT) es necesario conocer el organismo que se va a modelar, en este caso el Physarum Polycephalum. Por lo tanto, en esta sección daremos una breve introducción al Physarum Polycephalum, así como sus características y propiedades.

2.2.1. Mixomiceto

Los mixomicetos, también conocidos como myxo-mycetes o mohos mucilaginosos, son un grupo de fascinantes organismos unicelulares que se encuentran en el reino Protista, más concretamente protistas ameboides o amobozoa. A pesar de su apariencia poco llamativa y su tamaño microscópico, los mixomicetos son organismos muy interesantes, por su ciclo de vida y su comportamiento biológico inusual.

A diferencia de las setas y otros hongos tradicionales, los mixomicetos no forman estructuras multicelulares visibles durante la mayor parte de su ciclo de vida. En cambio, existen como células individuales, generalmente microscópicas, denominadas myxamoebas, que se desplazan a través de ambientes húmedos y ricos en materia orgánica, buscando condiciones favorables para su crecimiento.[13]

Los mixomicetos según Rojas [13] toman 3 formas distintas durante el transcurso de su vida:

- **Amoeboides:** Son células individuales que se mueven por medio de pseudópodos o flagelos dependiendo principalmente de la cantidad de agua en el medio. Estas amebas se denominan *myxamoebas* y son las que se encuentran en el suelo.
- **Plasmodio:** Es una masa de citoplasma multinucleado sin separación de membranas celulares, que se mueve por medio de la contracción de sus fibras de actina. Este plasmodio es el que se encuentra en el interior de los troncos de los árboles.
- **Cuerpo fructífero:** Es la estructura que se forma cuando el plasmodio se transforma en esporas. Estas esporas son las que se encuentran en la parte superior de los troncos de los árboles.

2.2.2. Ciclo de vida

El ciclo de vida de los mixomicetos es muy complejo, sin embargo, un mixomiceto típico tiene un ciclo de vida que se puede dividir en dos etapas distintas, las cuales son: el plasmodio y uno o más cuerpos fructíferos.[14]

La secuencia de eventos que ocurren durante el ciclo de vida de un mixomiceto típico comienza con una espora microscópica que se formó dentro de uno de los cuerpos fructíferos, característicamente producidos por los mixomicetos, y luego es liberada de este. Bajo condiciones favorables, la espora germina para producir de uno a cuatro protoplastos haploides sin pared celular, denominados gametos. Estos son liberados a través de un pequeño poro que se forma en la pared de la espora aunque también pueden resultar de la espora abriéndose.

Algunos protoplastos son flagelados cuando se liberan, mientras que otros son ameboides. Estos últimos a veces pueden desarrollar flagelos después de un corto período de tiempo o, en algunos casos, simplemente permanecer ameboides. Las células flageladas se llaman células enjambre, mientras que las células no flageladas se llaman mixoamoebas. Las mixoamoebas y las células enjambre son interconvertibles, y la forma particular en la que existe una célula dada depende aparentemente en gran medida de la disponibilidad de agua libre en su entorno inmediato. En presencia de agua libre, la forma flagelada tiende a predominar, mientras que bajo condiciones de escasez de agua, las mixamoebas sin flagelos son más comunes.[14]

Las mixamoebas y las células enjambre se dividen en dos, y cuando las condiciones no son buenas, las mixamoebas pueden convertirse en estructuras inactivas llamadas microquistes, que les ayudan a sobrevivir por mucho tiempo. Cuando dos de estas células se juntan, forman un cigoto, que puede ser ameboides o flagelado, pero eventualmente se convierte en ameboides y crece para convertirse en un plasmodio, una estructura grande con muchos núcleos pero que funciona como una sola célula. Si las condiciones se vuelven difíciles, como por la falta de agua o frío, el plasmodio se puede transformar en un esclerocio, una forma resistente que puede volver a convertirse en plasmodio cuando las condiciones mejoren. En invierno, es posible encontrar esclerocios debajo de la corteza de troncos y tocones en descomposición.

El ciclo de vida de los mixomicetos se puede observar en la Figura 11, en donde:

- A) Espora
- B) Germinación de la espora
- C) Etapa unicelular, que es o una mixoamiba (izquierda) o una célula enjambre (derecha).
- D) Microquiste
- E) Fusión de dos mixoamoebas o células enjambre para producir solo una célula.
- F) Fusión de dos mixoamoebas o células enjambre para producir solo una célula.
- G) Cigoto
- H) Plasmodio temprano
- I) Esclerocio
- J) Plasmodio maduro
- K) Comienzo de la esporulación
- L) Cuerpos fructíferos maduros con esporas aún encerradas

Si desea profundizar en el tema, puede consultar el libro "Myxomycetes: Biology, Systematics, Biogeography, and Ecology" de Rojas [13].

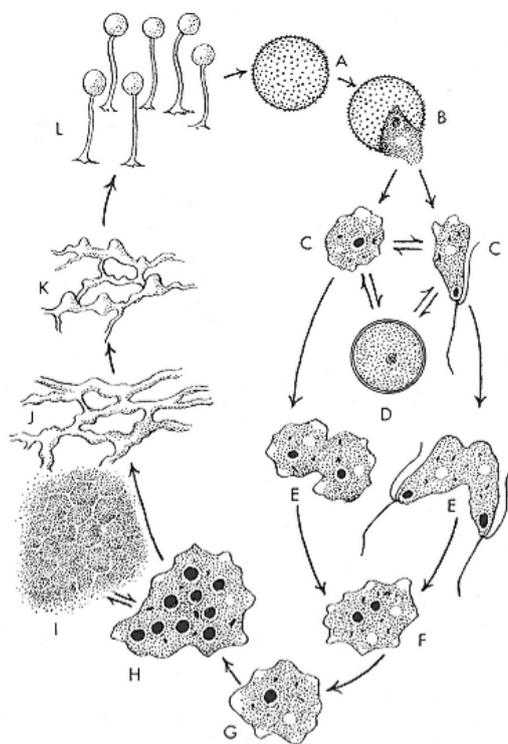


Figura 11: Ciclo de vida de los mixomicetos, extraído de "Myxomycetes: A Handbook of Slime Molds" de Stephenson y Stempen [14].

2.2.3. Physarum Polycephalum

El Physarum Polycephalum, también conocido como "The Blob", o "La Mancha", es un protista con formas celulares diversas. El Physarum Polycephalum es un mixomiceto acelular, esto proviene de la etapa plasmoidal de su ciclo de vida, en la cual el plasmodio es un coenocito multinucleado macroscópico de color amarillo brillante, formado en una red de tubos entrelazados. Esta etapa del ciclo de vida es la que se utiliza para el estudio de este organismo.[15] Podemos ver un ejemplo en la siguiente Figura 12.

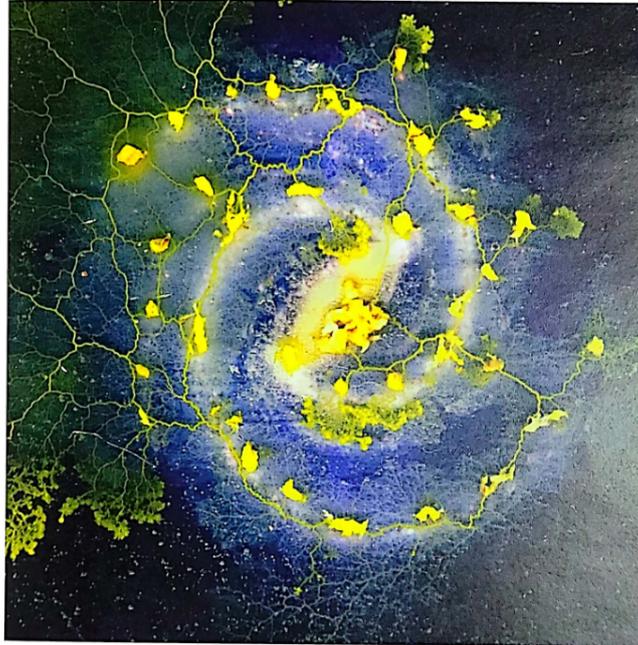


Figura 12: Physarum propagándose en una impresión artística de una galaxia. Imagen extraída de 'Atlas of Physarum Computing' de A. Adamatzky [16].

Como vimos con anterioridad, los mixomicetos se dividen en dos etapas, el plasmodio y los cuerpos fructíferos. El Physarum Polycephalum es un mixomiceto que se encuentra en la etapa plasmoidal de su ciclo de vida, en la cual el plasmodio es un coenocito multinucleado macroscópico de color amarillo brillante, formado en una red de tubos entrelazados. Esta etapa del ciclo de vida es la que se utiliza para el estudio de este organismo.[15]

El Physarum Polycephalum es un organismo que se encuentra en la naturaleza en lugares húmedos y oscuros, como en el interior de los troncos de los árboles en descomposición, en hojarasca húmeda, en suelos ricos en materia orgánica y en lugares oscuros y húmedos. Este organismo se alimenta de bacterias, hongos y otros microorganismos que se encuentran en su entorno, y se desplaza por medio de la contracción de sus fibras de actina, que le permiten moverse en busca de alimento.[15]

El Physarum Polycephalum es un organismo muy interesante para el estudio de la biología y la física, ya que tiene propiedades únicas que lo hacen un organismo muy especial. Por ejemplo, el Physarum Polycephalum es capaz de resolver laberintos como se observa en la Figura 13, encontrar la ruta más corta entre dos puntos, y tomar decisiones complejas basadas en la informa-



Figura 13: Physarum Polycephalum resolviendo un laberinto. [16].

ción que recibe de su entorno. Además, el Physarum Polycephalum es capaz de aprender y recordar información, y de adaptarse a su entorno de una manera muy eficiente.

Una vez dada una breve introducción al Physarum Polycephalum, podemos pasar a la perspectiva de la computación, en donde el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización, simulación de redes de transporte, y modelado de sistemas complejos. En la siguiente sección veremos cómo el Physarum Polycephalum ha sido utilizado en la computación y en la modelación de sistemas complejos.

2.2.4. El Physarum Polycephalum visto desde la perspectiva computacional

Como se mencionó en la sección 2.2.3, el Physarum Polycephalum es un organismo notable que ha despertado un gran interés por parte de biólogos y matemáticos debido a su notable capacidad para exhibir comportamientos emergentes y resolver problemas de optimización de manera eficiente, demostrando una gran versatilidad. Entre sus comportamientos complejos se encuentran la locomoción, la formación de redes adaptativas y la toma de decisiones descentralizadas.

En la computación, el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización, simulación de redes de transporte, y modelado de sistemas complejos. En particular, el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización de rutas, como el problema del camino más corto, el problema del flujo máximo, y el problema de la cobertura de sensores. Además, el Physarum Polycephalum ha sido utilizado para modelar sistemas complejos, como la formación de redes de transporte, la formación de patrones en sistemas biológicos, y la formación de estructuras en sistemas físicos.

Por mencionar algunos ejemplos de aplicaciones del Physarum Polycephalum en la computación, tenemos los siguientes:

- **A physarum-inspired prize-collecting steiner tree approach to identify subnetworks for drug repositioning:** En el artículo se detalla cómo un algoritmo, inspirado en el moho Physarum polycephalum, se aplica para descubrir medicamentos que podrían ser útiles en el tratamiento de enfermedades cardiovasculares. Mediante la construcción de Redes de Similitud de Fármacos (Drug Similarity Networks, DSNs), donde los nodos representan medicamentos y las conexiones reflejan similitudes entre ellos basadas en características como la estructura química y los efectos terapéuticos, cada medicamento recibe un 'premio' según su similitud con otros ya utilizados en afecciones cardiovasculares. El algoritmo busca dentro de estas redes para encontrar subredes que maximicen estos premios y minimicen los costos (disimilitudes), identificando así grupos de fármacos potencialmente reutilizables para tratar enfermedades cardiovasculares. Este método propone una forma innovadora de repensar el uso de medicamentos existentes, ofreciendo un camino acelerado hacia el descubrimiento de nuevas aplicaciones terapéuticas en el campo cardiovascular. [17]
- **A Novel Physarum-Based Ant Colony System for Solving the Real-World Traveling Salesman Problem:** Este artículo introduce un nuevo sistema de colonia de hormigas, inspirado en el modelo matemático de Physarum, para abordar el Problema del agente viajero (Traveling Salesman Problem, TSP). Este sistema ha demostrado ser más eficiente y robusto en comparación con los sistemas tradicionales de colonia de hormigas, algoritmos genéticos y optimización por enjambre de partículas. Este estudio se encuentra en un capítulo del libro 'Advances in Swarm Intelligence'. [18]

- **Composing Popular Music with Physarum polycephalum-based Memristors:** Este artículo investiga el uso de *Physarum polycephalum*, un moho mucilaginoso, como memristor para la composición de música popular, presentando una colaboración entre organismos biológicos y sistemas computacionales en la creación musical. Mediante una interfaz hardware-software, el estudio transforma datos musicales en voltajes y viceversa, utilizando el comportamiento no lineal del moho para influir en la composición. Aunque requiere ajustes para integrar las salidas del organismo en las piezas musicales, este enfoque innovador abre nuevas posibilidades en la creatividad computacional y la producción musical, instando a músicos y no expertos a explorar el cómputo no convencional en sus procesos creativos. El trabajo subraya el potencial de incorporar tecnologías biológicas en la composición musical, marcando un paso hacia la diversificación de las herramientas creativas en la música popular. [19]
- **Monte Carlo Physarum Machine: Characteristics of Pattern Formation in Continuous Stochastic Transport Networks:** El artículo introduce la Máquina de Physarum Monte Carlo (Monte Carlo Physarum Machine, MCPM), un modelo avanzado para reconstruir redes de transporte a partir de datos en 2D y 3D, ampliando un modelo previo de Jones sobre el moho *Physarum polycephalum*. La MCPM se evalúa por su capacidad para generar estructuras complejas denominadas poliformas y se aplica en la reconstrucción de la red cósmica, mostrando eficacia con datos cosmológicos simulados y observacionales. Los autores, afiliados a la Universidad de California, Santa Cruz y la Universidad Estatal de Nuevo México, exploran también aplicaciones futuras del MCPM en diversas disciplinas. [20]
- **Using an Artificial Physarum polycephalum Colony for Threshold Image Segmentation:** Este artículo presenta un innovador algoritmo basado en la simulación de una colonia de *Physarum polycephalum* artificial para abordar el problema de la segmentación de imágenes por umbral, un área clave en el procesamiento de imágenes. Tradicionalmente, los algoritmos de Inteligencia Artificial (IA) enfrentan desafíos en la selección del umbral óptimo, tendiendo a caer en óptimos locales. La metodología propuesta simula la expansión y contracción de hifas artificiales para buscar soluciones óptimas, facilitando el aprendizaje mutuo entre diferentes *Physarum polycephalum* y mejorando la capacidad de búsqueda global. Utilizando la entropía de Kapur como función de ajuste, el algoritmo propuesto demuestra una mayor precisión y velocidad de convergencia en comparación con métodos convencionales, validado a través de experimentos de referencia. Este enfoque abre nuevas perspectivas en el campo del procesamiento de imágenes, particularmente en aplicaciones de segmentación por umbral, ofreciendo una herramienta prometedora para resolver problemas complejos en esta área. [21]

Como se puede observar, el *Physarum Polycephalum* ha demostrado ser una fuente de inspiración para el desarrollo de algoritmos y sistemas computacionales innovadores, que han sido aplicados en una amplia variedad de campos, desde la biología y la medicina, hasta la música y la cosmología. Su capacidad para resolver problemas complejos de manera eficiente y su versatilidad para adaptarse a diferentes entornos lo convierten en un organismo único y valioso para la investigación científica y la computación.

2.3. Modo gráfico

Para poder desarrollar el simulador de *Physarum Polycephalum*, es necesario conocer el modo gráfico, ya que es la interfaz que el usuario va a utilizar para interactuar con el simulador.

2.3.1. Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)

Para el desarrollo de nuestro Trabajo Terminal (TT), se utilizó la Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML), la cual es una biblioteca gráfica multiplataforma de código abierto, que proporciona una Interfaz de Programación de Aplicaciones (Application Programming Interface, API) simple y fácil de usar para el desarrollo de aplicaciones multimedia y videojuegos. SFML está escrita en C++ y proporciona una interfaz de programación orientada a objetos, que facilita la creación de aplicaciones gráficas y multimedia de alto rendimiento.

En nuestro caso lo estamos usando en C++, ya que buscamos un mejor rendimiento en tiempo de ejecución la comparación la podemos ver en la Figura 14. Además de que SFML es una biblioteca muy popular en la comunidad de desarrollo de videojuegos, por lo que es una buena opción para el desarrollo de nuestro simulador. Y no es tan complicada como Vulkan o DirectX, y ya que el propósito de nuestro trabajo no es el desarrollo del simulador sino del Robot, no lo consideramos vital el uso de una biblioteca más compleja.

SFML proporciona una serie de módulos que permiten el desarrollo de aplicaciones multimedia y videojuegos, incluyendo gráficos 2D, sonido, entrada de teclado y ratón, y redes. Además, SFML proporciona una serie de clases y funciones que facilitan la creación de aplicaciones gráficas y multimedia, como ventanas, sprites, texturas, fuentes, sonidos, y eventos. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

SFML es una biblioteca multiplataforma que está disponible para Windows, Linux y macOS, y es compatible con una amplia variedad de compiladores y entornos de desarrollo, como Visual Studio, Code::Blocks, y Xcode. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

SFML es una biblioteca de código abierto que está disponible bajo la licencia zlib/png, lo que significa que se puede utilizar de forma gratuita en proyectos comerciales y no comerciales, y se puede modificar y distribuir libremente. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

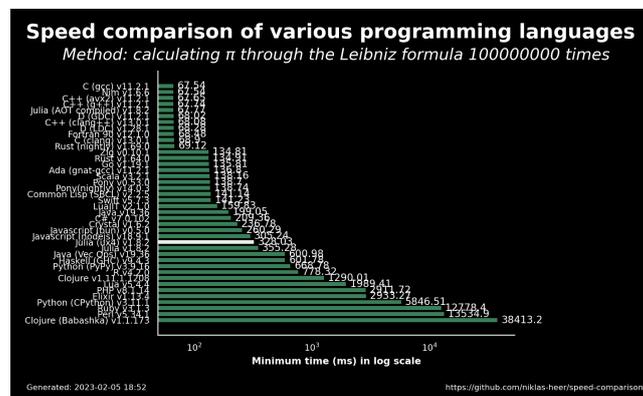


Figura 14: Comparación de tiempo de ejecución entre diferentes lenguajes de Programación. Esta gráfica fue generada Heer [22]

2.4. Raspberry Pi

En nuestro Trabajo Terminal (TT), la Raspberry Pi 4 es la encargada de gestionar el robot y la ruta que nos da el Physarum por medio de bluetooth. Por ello en esta sub sección, daremos una breve introducción a la Raspberry Pi 4, especificaciones técnicas, comparativas, etc.

2.4.1. Historia y evolución

La Raspberry Pi nació en 2006 como un proyecto ideado por Eben Upton, Rob Mullins, Jack Lang y Alan Mycroft, quienes trabajaban en la Universidad de Cambridge. La idea principal era crear una computadora de bajo costo que permitiera a los estudiantes de la universidad mejorar sus habilidades de programación. [23] En 2009, el proyecto se convirtió en una fundación sin fines de lucro, la Raspberry Pi Foundation, con el objetivo de promover la enseñanza de la informática en las escuelas y países en desarrollo. [23]

La primera Raspberry Pi fue lanzada en febrero de 2012, con un procesador ARM11 de 700 MHz, 512 MB de Memoria de Acceso Aleatorio (Random Access Memory, RAM) y un precio de 35 dólares. Desde entonces, la Raspberry Pi ha evolucionado hasta convertirse en una plataforma de desarrollo muy popular, con millones de unidades vendidas en todo el mundo.[23] La Raspberry Pi 4, lanzada en junio de 2019, es la versión más reciente de la placa y cuenta con un procesador ARM Cortex-A72 de 1.5 GHz, hasta 8 GB de RAM y soporte para pantallas 4K. [23]

La Raspberry Pi ha sido utilizada en una amplia variedad de proyectos, desde servidores web y centros multimedia hasta robots y sistemas de control. Su bajo costo y su flexibilidad la han convertido en una herramienta muy popular entre los aficionados a la informática y la electrónica. Además, la Raspberry Pi ha sido utilizada en proyectos educativos en todo el mundo, ayudando a enseñar a los jóvenes las habilidades necesarias para el siglo XXI.

- **Raspberry Pi Model B:** La Raspberry Pi Model B es la primera versión de la placa, lanzada en febrero de 2012. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 1 puerto USB tipo A, 1 conector Entrada/Salida de Propósito General (General Purpose Input/Output, GPIO) de 8 pines, salida HDMI, salida de audio y un lector de tarjetas SD [23]
- **Raspberry Pi Model A+:** La Raspberry Pi Model A+ es una versión más pequeña y económica de la placa, lanzada en noviembre de 2014. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 1 puerto USB tipo A, 1 conector GPIO de 40 pines, salida HDMI y salida de audio 3.5 mm [23]
- **Raspberry Pi 2 Model B:** La Raspberry Pi 2 Model B es la segunda versión de la placa, lanzada en febrero de 2015. Cuenta con un procesador ARM Cortex-A7 de 900 MHz, 1 GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm y ethernet 10/100 [23]
- **Raspberry Pi Zero:** La Raspberry Pi Zero es una versión más pequeña y económica de la placa, lanzada en noviembre de 2015. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines y HAT compatible de 40 pines [23]
- **Raspberry Pi 3 Model B:** La Raspberry Pi 3 Model B es la tercera versión de la placa, lanzada en febrero de 2016. Cuenta con un procesador ARM Cortex-A53 de 1.2 GHz, 1

GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, ethernet 10/100, conexión Wifi y Bluetooth 4.1 LE [23]

- **Raspberry Pi Zero W:** La Raspberry Pi Zero W es una versión más pequeña y económica de la placa, lanzada en febrero de 2017. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluetooth 4.1 LE [23]
- **Raspberry Pi Zero WH:** La Raspberry Pi Zero WH es una versión más pequeña y económica de la placa, lanzada en febrero de 2018. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluetooth 4.1 LE [23]
- **Raspberry Pi 3 Model B+:** La Raspberry Pi 3 Model B+ es la cuarta versión de la placa, lanzada en marzo de 2018. Cuenta con un procesador ARM Cortex-A53 de 1.4 GHz, 1 GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, ethernet 10/100, conexión Wifi y Bluetooth 4.2 LE [23]
- **Raspberry Pi 3 Model A+:** La Raspberry Pi 3 Model A+ es una versión más pequeña y económica de la placa, lanzada en noviembre de 2018. Cuenta con un procesador ARM Cortex-A53 de 1.4 GHz, 512 MB de RAM, 1 puerto USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, conexión Wifi y Bluetooth 4.2 LE [23]
- **Raspberry Pi 4 Model B:** La Raspberry Pi 4 Model B es la quinta versión de la placa, lanzada en junio de 2019. Cuenta con un procesador ARM Cortex-A72 de 1.5 GHz, hasta 8 GB de RAM, 2 puertos USB tipo A 3.0, 2 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, 2 salidas micro HDMI, salida de audio 3.5mm, ethernet Gigabit, conexión Wifi y Bluetooth 5.0 LE [23]
- **Raspberry Pi Compute Module 1:** La Raspberry Pi Compute Module 1 es una versión de la placa diseñada para su uso en sistemas embebidos, lanzada en abril de 2014. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 4GB eMMC Flash, Conector SODIMM DDR2 [23]
- **Raspberry Pi Compute Module 3:** La Raspberry Pi Compute Module 3 es una versión de la placa, lanzada en enero de 2017. Cuenta con un procesador BCM2837 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, 4GB eMMC Flash, Conector SODIMM DDR2 y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 3 Lite:** La Raspberry Pi Compute Module 3 Lite es una versión de la placa, lanzada en enero de 2017. Cuenta con un procesador BCM2837 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, Conector SODIMM DDR2 y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 3+:** La Raspberry Pi Compute Module 3+ es una versión de la placa, lanzada en enero de 2019. Cuenta con un procesador BCM2837B0 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, 8GB, 16GB y 32 GB eMMC Flash, slot MicroSDHC y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 4:** La Raspberry Pi Compute Module 4 es una versión de la placa, lanzada en octubre de 2020. Cuenta con un procesador ARM a 1.5 GHz, 1 GB, 2 GB, 4 GB, 8 GB de RAM, 2 puertos Gigabit Ethernet, Conectividad Wi-Fi (opcional), 1 USB C y conector GPIO de 28 pines [23]

- **Raspberry Pi 400:** La Raspberry Pi 400 es una versión de la placa, lanzada en noviembre de 2020. Cuenta con un procesador ARM Cortex-A72 de 1.5 GHz y soporte de 64 bits, 1-8 GB de RAM, 2 puertos USB tipo A 3.0, 2 puertos USB tipo A 2.0, Conector GPIO de 40 pines, 2 salidas micro HDMI, salida de audio 3.5mm, ethernet Gigabit, conexión Wifi y Bluetooth 5.0 LE [23]
- **Raspberry Pi Pico:** La Raspberry Pi Pico es una placa de desarrollo, lanzada en enero de 2021. Cuenta con un procesador RP2040 de doble núcleo ARM Cortex-M0+ a 133 MHz, 264 KB de RAM, 2 MB de memoria flash QSPI, 26 pines GPIO, 3 pines analógicos, 2 Transmisor-Receptor Asíncrono Universal (Universal Asynchronous Receiver-Transmitter, UART), 2 SPI, 2 I2C, 16 canales Modulación por Ancho de Pulso (Pulse Width Modulation, PWM), 1 temporizador de 12 bits y 1 temporizador de 16 bits [23]
- **Raspberry Pi Zero 2 W:** La Raspberry Pi Zero 2 W es una versión de la placa, lanzada en octubre de 2021. Cuenta con un procesador BCM2710A1 de cuatro núcleos a 1.0 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluetooth 4.2 LE [23]
- **Raspberry Pi 5:** La Raspberry Pi 5 es una versión de la placa, lanzada en octubre de 2023. Cuenta con un procesador ARM Cortex-A73 de 2.4 GHz, hasta 8 GB de RAM, Doble salida micro HDMI 4K60p, gpu VideoCore VII con soporte de OpenGL ES 2.1 y Vulkan 1.2, decodificador HEVC 4K60, Bluetooth 5.0, WiFi 802.11ac, Ranura microSD de alta velocidad con soporte de SDR104, 2 puertos USB 3.0, 2 puertos USB 2.0, 1 puerto Gigabit Ethernet, interfaz PCIe 2.0, conexiones GPIO de 40 pines y botón de encendido y apagado [23]

2.4.2. Comparativa

Una vez visto los modelos de Raspberry Pi, es necesario hacer una comparativa entre ellos para poder elegir el modelo que mejor se adapte a nuestras necesidades. En el Cuadro 2 se muestra una comparativa entre los modelos de Raspberry Pi.

Cuadro 2: Comparación de los modelos de Raspberry Pi

Modelo	CPU	RAM	Puertos USB	GPIO	Salida Video	Red	Memoria
Model B	ARM11 700 MHz	512 MB	1 tipo A	8 pines	HDMI	No	Lector SD
Model A+	ARM11 700 MHz	512 MB	1 tipo A	40 pines	HDMI, Audio 3.5mm	No	No
2 Model B	ARM Cortex-A7 900 MHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100	No
Zero	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	No	No
3 Model B	ARM Cortex-A53 1.2 GHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100, Wifi, BT 4.1	No

Continúa en la siguiente página

Cuadro 2 – continuación de la página anterior

Modelo	CPU	RAM	Puertos USB	GPIO	Salida Video	Red	Memoria
Zero W	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.1	No
Zero WH	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.1	No
3 Model B+	ARM Cortex-A53 1.4 GHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100, Wifi, BT 4.2	No
3 Model A+	ARM Cortex-A53 1.4 GHz	512 MB	1 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Wifi, BT 4.2	No
4 Model B	ARM Cortex-A72 1.5 GHz	hasta 8 GB	2 tipo A 3.0, 2 tipo A 2.0	40 pines	2 micro HDMI	Ethernet Gigabit, Wifi, BT 5.0	MicroSD
Compute Module 1	ARM11 700 MHz	512 MB	No	SODIMM DDR2	No	No	4GB eMMC
Compute Module 3	BCM2837 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	4GB eMMC
Compute Module 3 Lite	BCM2837 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	No
Compute Module 3+	BCM2837B0 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	8/16/32 GB eMMC
Compute Module 4	ARM 1.5 GHz	1/2/4/8 GB	USB C	GPIO 28 pines	No	2x Ethernet Gigabit, Wifi (opc)	No
Pi 400	ARM Cortex-A72 1.5 GHz	1-8 GB	2 tipo A 3.0, 2 tipo A 2.0	40 pines	2 micro HDMI	Ethernet Gigabit, Wifi, BT 5.0	No
Pico	RP2040 ARM Cortex-M0+ 133 MHz	264 KB	No	26 GPIO	No	No	2 MB flash
Zero 2 W	BCM2710A1 1.0 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.2	No
Pi 5	ARM Cortex-A73 2.4 GHz	hasta 8 GB	2 USB 3.0, 2 USB 2.0	40 pines	Doble micro HDMI 4K60p	Ethernet Gigabit, Wifi, BT 5.0	MicroSD SDR104

2.4.3. Raspberry Pi 4 Model B

Como ya vimos con anterioridad en la sección 2.4.2, la Raspberry Pi 4 Model B es una computadora de placa única (Single Board Computer, SBC) desarrollada por la Fundación Raspberry Pi. Es la cuarta generación de la serie Raspberry Pi y fue lanzada en junio de 2019. Nosotros enfatizaremos sus usos que tienen para el desarrollo de un robot autónomo, como el que estamos desarrollando en nuestro Trabajo Terminal (TT).

Ventajas de la Raspberry Pi 4 Model B:

- **Alto Rendimiento:** El procesador Broadcom BCM2711 de cuatro núcleos a 1.5GHz ofrece un rendimiento significativamente mayor que las versiones anteriores de la Raspberry Pi, lo que permite ejecutar algoritmos de control y navegación más complejos.
- **Conectividad Avanzada:** La Raspberry Pi 4 Model B cuenta con dos puertos USB 3.0, dos puertos USB 2.0, un puerto Gigabit Ethernet, conexión Wi-Fi 802.11ac y Bluetooth 5.0, lo que facilita la comunicación con otros dispositivos, sensores y redes.
- **Flexibilidad de E/S:** La placa ofrece una amplia variedad de opciones de entrada y salida, incluyendo Entrada/Salida de Propósito General (General Purpose Input/Output, GPIO), HDMI, USB, Ethernet, Wi-Fi, Bluetooth, cámaras y pantallas, lo que permite conectar una gran variedad de sensores, actuadores y periféricos.
- **Soporte de Software:** La Raspberry Pi 4 Model B es compatible con una amplia variedad de sistemas operativos, incluyendo Raspbian, Ubuntu, Windows 10 IoT Core y otros, lo que facilita el desarrollo de aplicaciones y la integración con otros dispositivos.
- **Bajo Costo:** La Raspberry Pi 4 Model B es una placa de bajo costo, lo que la hace accesible para estudiantes, aficionados y profesionales que deseen desarrollar proyectos de robótica y automatización.

Usos de la Raspberry Pi 4 Model B en robótica:

- **Control de Robots:** La Raspberry Pi 4 Model B se puede utilizar para controlar robots móviles, drones, brazos robóticos y otros dispositivos autónomos, gracias a su alto rendimiento, conectividad avanzada y flexibilidad de E/S.
- **Visión por Computadora:** La Raspberry Pi 4 Model B se puede utilizar para procesar imágenes y vídeos en tiempo real, lo que permite a los robots detectar objetos, seguir líneas, evitar obstáculos y realizar otras tareas de visión por computadora.
- **Aprendizaje Automático:** La Raspberry Pi 4 Model B se puede utilizar para ejecutar algoritmos de aprendizaje automático y redes neuronales, lo que permite a los robots aprender de su entorno, adaptarse a nuevas situaciones y mejorar su rendimiento con el tiempo.
- **Interacción con el Entorno:** La Raspberry Pi 4 Model B se puede utilizar para interactuar con el entorno físico a través de sensores y actuadores, lo que permite a los robots medir la temperatura, la humedad, la luz, la distancia, la velocidad y otras variables, así como controlar motores, luces, pantallas y otros dispositivos.
- **Comunicación Inalámbrica:** La Raspberry Pi 4 Model B se puede utilizar para comunicarse de forma inalámbrica con otros dispositivos, sensores y redes, lo que permite a los robots enviar y recibir datos, comandos y actualizaciones de forma remota.

Por todo lo anteriormente mencionado, la Raspberry Pi 4 Model B es una excelente opción para el desarrollo de un robot autónomo, ya que ofrece un alto rendimiento, conectividad avanzada, flexibilidad de E/S, soporte de software y bajo costo, lo que la hace accesible para estudiantes, aficionados y profesionales que deseen desarrollar proyectos de robótica y automatización.

2.5. Protocolos de comunicación

La comunicación entre dispositivos es un aspecto fundamental en la robótica, ya que permite la interacción entre los diferentes componentes de un sistema. En nuestro Trabajo Terminal (TT), la comunicación entre el robot y el Physarum se realiza por medio de una aplicación móvil que se comunica mediante diversos protocolos de comunicación. En esta subsección, se describirán los protocolos de comunicación utilizados en nuestro Trabajo Terminal (TT).

2.5.1. WebSocket

El protocolo de WebSocket fue desarrollado ya que el protocolo HTTP no es adecuado para aplicaciones en tiempo real, esto por que el protocolo HTTP es de petición-respuesta, lo que significa que el cliente debe solicitar información al servidor y el servidor debe responder a la solicitud. En cambio, el protocolo WebSocket permite una comunicación bidireccional entre el cliente y el servidor, en otras palabras, existe una conexión persistente entre el cliente y el servidor, lo que permite que el servidor envíe información al cliente sin que este lo solicite. [24]

A diferencia de los protocolos de comunicación tradicionales, WebSocket permite reducir la latencia en aplicaciones que requieren una actualización constante de datos, como juegos multijugador, chats en tiempo real, o plataformas de trading financiero. Esta capacidad es posible gracias al establecimiento de una conexión persistente a través de un único canal Protocolo de Control de Transmisión (Transmission Control Protocol, TCP), que permanece abierta hasta que alguna de las partes decide cerrarla. Esto reduce la sobrecarga de establecer conexiones repetidas y mejora significativamente el rendimiento de las aplicaciones que requieren actualizaciones constantes. [25]

El proceso de establecimiento de una conexión WebSocket comienza con un "handshake" basado en HTTP, donde el cliente solicita la apertura de una conexión WebSocket al servidor utilizando un encabezado específico, y el servidor responde aceptando o rechazando la conexión. Una vez completado el "handshake", la conexión se actualiza y ambos pueden intercambiar mensajes en formato binario o texto sin necesidad de seguir el ciclo de solicitud-respuesta. Esto hace que WebSocket sea altamente eficiente para aplicaciones en tiempo real que manejan grandes cantidades de datos o requieren baja latencia. [26]

Además, WebSocket proporciona ventajas en cuanto a la reducción del uso de ancho de banda. Al evitar la necesidad de crear múltiples conexiones y al eliminar los encabezados HTTP innecesarios en cada intercambio de mensajes, se logra una transmisión de datos más ligera. Esto es especialmente útil en entornos donde los recursos de red son limitados, como dispositivos móviles o redes con baja velocidad. [27]

Sin embargo, aunque WebSocket ofrece muchas ventajas en términos de rendimiento y latencia, su implementación puede presentar desafíos de seguridad, como la exposición a ataques de secuestro de WebSocket entre sitios (Cross-Site WebSocket Hijacking, CSWSH) o vulnerabilidades de inyección. Por esta razón, es importante integrar medidas de seguridad, como el uso de WebSockets sobre el protocolo de Seguridad de la Capa de Transporte (Transport Layer

Security, TLS), conocido como WSS para cifrar las comunicaciones, y políticas adecuadas de validación del origen de las conexiones [28].

2.5.2. Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP)

El protocolo Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP) es un protocolo de comunicación utilizado en la World Wide Web para la transferencia de información entre un cliente y un servidor. Fue diseñado para ser un protocolo simple y flexible, que permitiera la transferencia de datos de manera eficiente y segura. [29]

HTTP opera bajo el modelo petición-respuesta, donde el cliente envía una petición al servidor solicitando un recurso específico, y el servidor responde con el recurso solicitado o un código de estado que indica si la petición fue exitosa o no. Las peticiones y respuestas en HTTP están compuestas por un conjunto de encabezados y opcionalmente un cuerpo de mensaje, que contiene la información a transferir. [29]

HTTP es un protocolo sin estado, lo que significa que cada petición se procesa de manera independiente, sin tener en cuenta las peticiones anteriores. Esto permite que el servidor sea más escalable y flexible, ya que no necesita mantener un estado de sesión con cada cliente. Sin embargo, esta característica también implica que el servidor no puede recordar información sobre el cliente entre peticiones, lo que puede limitar la interacción entre el cliente y el servidor. [29]

HTTP utiliza el Protocolo de Control de Transmisión (Transmission Control Protocol, TCP) como su capa de transporte, lo que garantiza una comunicación fiable y ordenada entre el cliente y el servidor. Las conexiones HTTP se establecen mediante un *handshake* entre el cliente y el servidor, donde se negocian los parámetros de la conexión, como el tipo de contenido aceptado, la codificación de transferencia, y la longitud del cuerpo del mensaje. Una vez establecida la conexión, el cliente y el servidor pueden intercambiar mensajes de manera eficiente y segura. [29]

A pesar de su simplicidad y flexibilidad, HTTP tiene algunas limitaciones en términos de rendimiento y eficiencia. Por ejemplo, HTTP es un protocolo de texto plano, lo que significa que los mensajes enviados a través de HTTP deben ser codificados en texto legible por humanos, lo que puede aumentar el tamaño de los mensajes y reducir la eficiencia de la transferencia de datos. Además, HTTP no es adecuado para aplicaciones en tiempo real, ya que su modelo petición-respuesta puede introducir latencia en la comunicación entre el cliente y el servidor. [29]

A pesar de estas limitaciones, HTTP sigue siendo uno de los protocolos de comunicación más utilizados en la World Wide Web, debido a su simplicidad, flexibilidad y compatibilidad con una amplia variedad de plataformas y tecnologías. Sin embargo, en aplicaciones que requieren una comunicación más eficiente y en tiempo real, es posible que sea necesario utilizar protocolos más especializados, como WebSocket o el Protocolo de Telemetría de Colas de Mensajes (Message Queuing Telemetry Transport, MQTT), que están diseñados específicamente para este propósito. [29]

2.5.3. Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC)

Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC) es un conjunto de tecnologías que permite la comunicación en tiempo real entre navegadores web y aplicaciones móviles. Fue desarrollado por Google en 2011 con el objetivo de facilitar la creación de aplicaciones de comunicación en tiempo real, como videollamadas, conferencias web y transmisión de datos en tiempo real. [30]

WebRTC se basa en varios estándares abiertos, como el Protocolo de Transporte en Tiempo Real (Real-Time Transport Protocol, RTP), el Protocolo de Control de Transporte en Tiempo Real (Real-Time Transport Control Protocol, RTCP) y Protocolo de Descripción de Sesión (Session Description Protocol, SDP), que permiten la transmisión de datos en tiempo real a través de la web. Estos estándares están diseñados para ser compatibles con una amplia variedad de dispositivos y plataformas, lo que facilita la creación de aplicaciones de comunicación en tiempo real que funcionan en diferentes entornos. [30]

Una de las características más importantes de WebRTC es su capacidad para establecer conexiones punto a punto entre los clientes, lo que permite una comunicación directa y segura entre los usuarios sin necesidad de pasar por un servidor centralizado. Esto reduce la latencia y mejora la calidad de la comunicación, ya que los datos se transmiten directamente entre los clientes sin intermediarios. Además, al utilizar cifrado de extremo a extremo, WebRTC garantiza la privacidad y seguridad de las comunicaciones, protegiendo los datos de posibles ataques o interceptaciones. [30]

WebRTC es compatible con una amplia variedad de dispositivos y plataformas, incluyendo navegadores web, aplicaciones móviles y dispositivos Internet de las Cosas (Internet of Things, IoT), lo que lo convierte en una solución versátil para la creación de aplicaciones de comunicación en tiempo real en diferentes entornos. Además, al ser un estándar abierto, WebRTC está respaldado por una amplia comunidad de desarrolladores y empresas, lo que garantiza su compatibilidad y soporte a largo plazo. [30]

3. Estado del Arte

En esta sección, presentamos un resumen de los trabajos previos relacionados con algoritmos previamente implementados del Physarum Polycephalum. Además, se incluyen trabajos relacionados con autómatas celulares y su aplicación en espacios euclidianos. Finalmente, se presentan trabajos relacionados al monitoreo de sistemas poblacionales y sistemas relacionados.

3.1. Physarum Polycephalum

En esta sección nos concentraremos principalmente en los diferentes modelos que se han propuesto para modelar el Physarum Polycephalum, así como en las aplicaciones que se han desarrollado a partir de estos modelos. Principalmente son 5 los modelos que se han propuesto para modelar el Physarum Polycephalum, los cuales son: el modelado de Adamatzky, el modelado de Olvera, el modelado de Marín, el modelado de Jones y el modelado de Gunji. A continuación, se describirán brevemente cada uno de estos modelos.

3.1.1. Modelado de Adamatzky

El modelo de *Physarum polycephalum* de Andrew Adamatzky destaca sus capacidades computacionales. Manipulando *Physarum* con alimentos y repelentes, demuestra la creación de puertas lógicas y la resolución de problemas de optimización como el camino más corto y el problema del agente viajero [31]. En la Figura 15 se muestra un ejemplo de puertas lógicas creadas por *Physarum*.

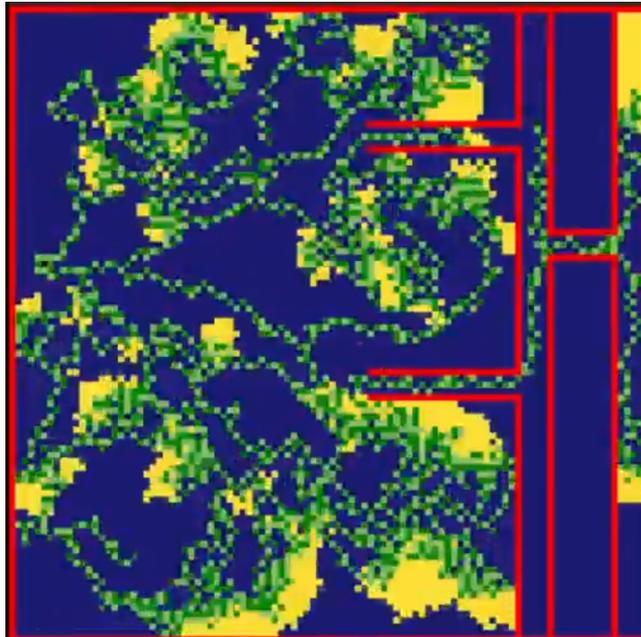


Figura 15: Ejemplo de puertas lógicas (OR) creadas por *Physarum*.

Adamatzky profundiza en la red protoplasmática de *Physarum*, comparándola con sistemas humanos, y muestra propiedades memristivas similares a los memristores electrónicos [31]. Su investigación también explora la dinámica no lineal y la formación de patrones complejos, significativos para la computación no convencional [31].

El modelo utiliza partículas de enjambre en un entorno bidimensional (2D) para simular el movimiento ameboide de *Physarum*. Estas partículas tienen etapas sensitiva y motora, interactuando con un quimioatrayente y generando patrones complejos. Además, se considera la resistencia al movimiento y la influencia de estímulos externos como quimioatrayentes y luz, controlando el comportamiento del colectivo.

Una característica destacada es la capacidad del colectivo para cambiar y recuperar su forma, navegar obstáculos y dividirse en fragmentos independientes que pueden fusionarse nuevamente, lo cual es deseable en aplicaciones robóticas [31]. La Figura 16 ilustra cómo el colectivo de *Physarum* se adapta a obstáculos.

Para más detalles, ver '*Emergence of self-organized amoeboid movement in a multi-agent approximation of physarum polycephalum*' [33].



Figura 16: Colectivo de *Physarum* adaptándose a obstáculos. [32]

3.1.2. Modelado Guillermo Olvera

El modelo utiliza autómatas celulares con la vecindad de Moore para simular la propagación y búsqueda de rutas del organismo *Physarum Polycephalum*. Este modelo funciona en una cuadrícula bidimensional donde cada célula puede asumir uno de varios estados: campo libre, nutriente no encontrado, repelente, punto inicial, gel en contracción, gel en compuesto, nutriente hallado, expansión del *Physarum* y gel sin compuesto.

El algoritmo está implementado en Python, lo cual introduce ciertas limitaciones en términos de velocidad y escalabilidad. Debido a la naturaleza interpretada de Python, el modelo es lento y poco escalable cuando se incrementa el número total de células e hilos utilizados.

El proceso comienza con la designación de una célula inicial que representa el punto de inicio del *Physarum*. Las reglas de transición determinan cómo cambian los estados de las células en función de sus vecinos. Por ejemplo, una célula de campo libre se convierte en una célula de expansión del *Physarum* si está adyacente a una célula en estado de punto inicial, gel en contracción o nutriente hallado. Las células de expansión del *Physarum* se propagan por la cuadrícula, y al encontrarse con nutrientes, estas células cambian su estado a nutriente hallado, lo que refuerza la ruta. En la Figura 17 se muestra la configuración inicial del *Physarum* en la cuadrícula.

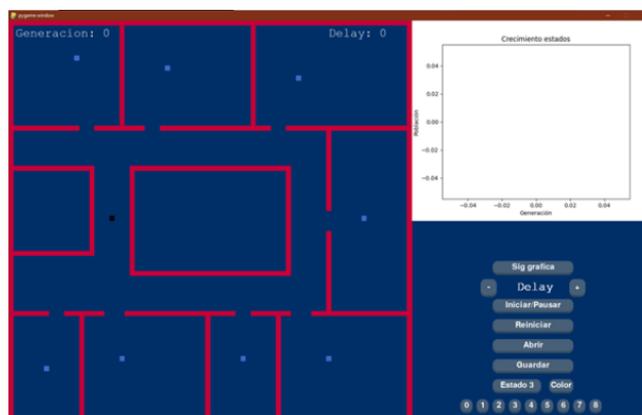


Figura 17: Configuración inicial del *Physarum* en la cuadrícula. [34]

A medida que el *Physarum* se expande, se generan rutas que conectan las fuentes de nutrientes, adaptándose dinámicamente a la presencia de obstáculos y asegurando la conectividad en el entorno. Aunque el algoritmo garantiza la formación de al menos una ruta viable, la trayectoria del *Physarum* no es realmente aleatoria, ya que sigue patrones determinados por las reglas de transición. Estas reglas, sin embargo, no siempre son claras ni consistentes. En la Figura 18 se muestra la expansión del *Physarum* y la formación de rutas.

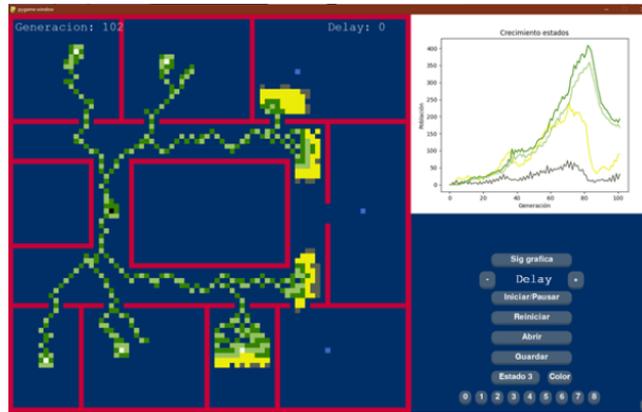


Figura 18: Expansión del *Physarum* y formación de rutas. [34]

Las reglas de transición se definen para cada estado de la célula, asegurando que el *Physarum* pueda encontrar y seguir rutas hacia los nutrientes, adaptándose en tiempo real a cambios en el entorno. Sin embargo, estas reglas no siempre son claras ni están bien documentadas en la referencia [34].

3.1.3. Modelado de Yair Marin

El algoritmo utilizado para modelar el comportamiento del *Physarum Polycephalum* en el documento se basa en autómatas celulares y se define por un conjunto de estados y reglas de transición específicas. Los estados incluyen campo libre, nutriente no encontrado, repelente, punto inicial, gel en contracción, gel con compuesto, nutriente hallado, expansión del *Physarum* y gel sin compuesto. Estos estados evolucionan de acuerdo con la vecindad de von Neumann, que considera las células adyacentes en las direcciones norte, sur, este y oeste.

Las reglas de transición determinan cómo cambia el estado de cada célula en función de sus vecinos. Por ejemplo, una célula en estado de campo libre (q_0) puede pasar al estado de expansión del *Physarum* (q_7) si está adyacente a un punto inicial (q_3), gel en contracción (q_4) o nutriente hallado (q_6). Del mismo modo, una célula en estado de nutriente no encontrado (q_1) cambia a estado de nutriente hallado (q_6) si está cerca de un gel con compuesto (q_5) o otro nutriente hallado (q_6). Estas reglas permiten que el modelo emule el comportamiento del *Physarum* en la búsqueda y exploración de su entorno, formando redes eficientes para el transporte de nutrientes y adaptándose a cambios en el entorno [35].

En la Figura 19, se muestra una ejecución del algoritmo, demostrando cómo el *Physarum* se expande y encuentra nutrientes en un entorno simulado.

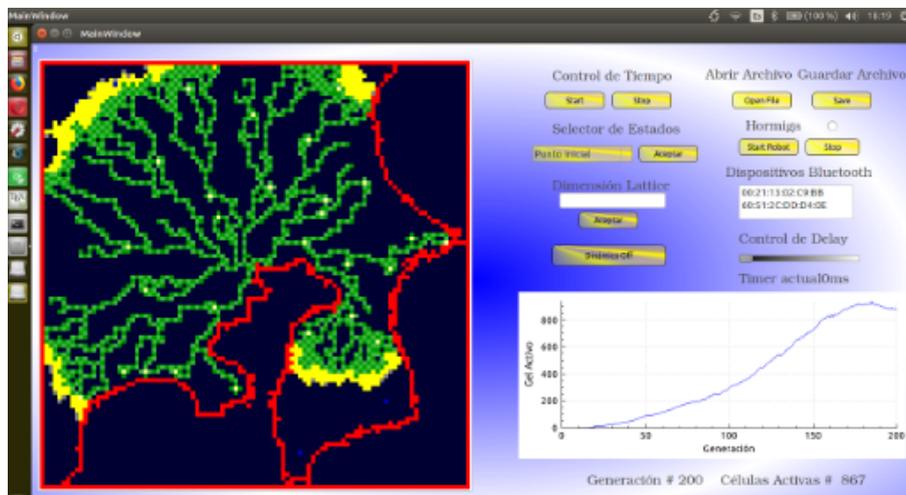


Figura 19: Ejecución del algoritmo de *Physarum Polycephalum* encontrando nutrientes. [35]

3.1.4. Modelado de Jeff Jones

El algoritmo propuesto en 'From Pattern Formation to Material Computation: Multi-agent Modelling of Physarum Polycephalum' [36] aprovecha el comportamiento natural de *Physarum polycephalum* para resolver problemas computacionales mediante un marco de sistema multi-agente (MAS). El núcleo del algoritmo involucra un gran número de agentes simples que imitan el comportamiento del plasmodio de *Physarum*. Cada agente opera basado en reglas locales, moviéndose e interactuando dentro de un entorno virtual que simula el espacio físico donde reside el moho del limo.

Los agentes se mueven hacia las fuentes de nutrientes siguiendo gradientes químicos, representando las señales atrayentes utilizadas por *Physarum*. Dejan rastros similares a feromonas que refuerzan los caminos exitosos, de manera similar a cómo *Physarum* fortalece sus tubos protoplasmáticos. Este mecanismo de retroalimentación de feromonas permite que los agentes se adapten dinámicamente a los cambios en el entorno, optimicen caminos y encuentren soluciones a problemas como el camino más corto o la resolución de laberintos. El comportamiento colectivo de estos agentes simples conduce a la emergencia de redes complejas y eficientes que pueden ser utilizadas para tareas de computación no convencional.

La fortaleza del algoritmo radica en su capacidad de autoorganización y adaptación sin control central. Demuestra capacidades robustas de resolución de problemas incluso en entornos dinámicos e inciertos. Al aprovechar los principios de autoorganización e interacción local, el algoritmo ofrece un enfoque novedoso para la computación distribuida y la optimización, inspirado en el comportamiento natural de *Physarum polycephalum*.

En la Figura 20 se muestra una representación del agente de acuerdo con el modelo de Jeff Jones.

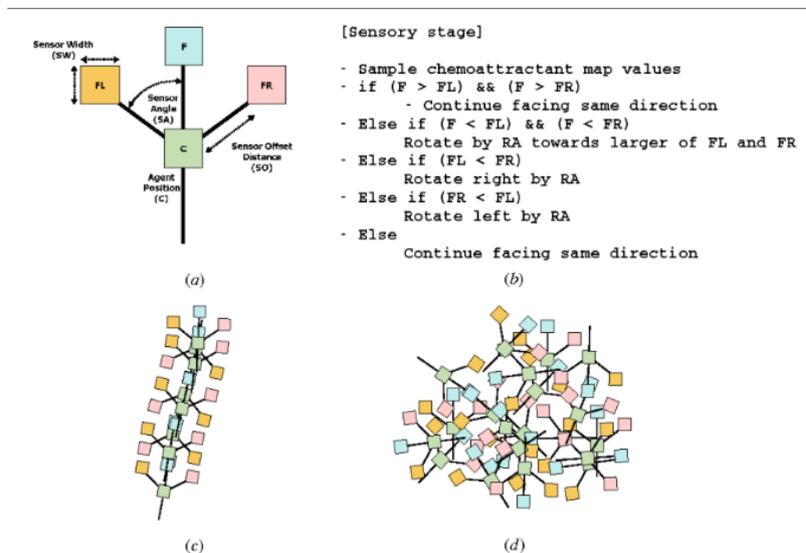


Figura 20: Representación del agente de acuerdo con el modelo de Jeff Jones. [36]

3.1.5. Modelado de Gunji

El modelo de célula mínima inspirado en el comportamiento del moho del limo *Physarum polycephalum* simula la capacidad de la célula para moverse y resolver problemas complejos como laberintos y configuraciones de árboles generadores a través de mecanismos simples pero efectivos. La célula está representada en una rejilla plana, donde cada sitio puede estar en uno de varios estados: externo (0), interno (1), límite (2) o estado final (-1). El modelo presenta dos fases principales: desarrollo y búsqueda de alimento. Durante la fase de desarrollo, la célula crece desde una semilla inicial hasta formar una agregación estructurada, mientras que en la fase de búsqueda de alimento, modifica activamente su forma y se mueve 'comiendo' sitios externos, causando flujo citoplasmático y reorganización de los límites.

Un aspecto clave del modelo es el proceso de 'comer 0', donde un sitio en estado 0 (externo) invade la célula, convirtiéndose en una 'burbuja' que es transportada dentro de la célula sin cruzar su propio camino (flujo memorizado). Este proceso conduce a la formación y eliminación de tentáculos, creación de redes adaptativas y optimización de caminos para resolver problemas como laberintos y configuraciones de árboles generadores. La interacción entre el flujo citoplasmático local y la forma global de la célula, impulsada por la alternancia entre endurecimiento y ablandamiento citoplasmático, permite que la célula se adapte dinámicamente y mantenga su estructura, exhibiendo comportamientos similares a la resolución inteligente de problemas observada en *Physarum polycephalum* [37].

En la Figura 21, se muestra cómo se aplica el algoritmo para resolver un laberinto. Este ejemplo ilustra la capacidad del modelo para adaptarse y optimizar caminos en tiempo real.

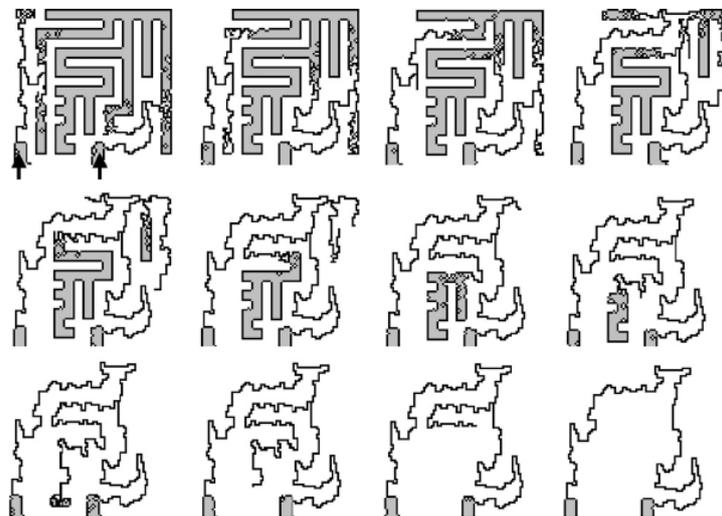


Figura 21: Aplicación del algoritmo para resolver un laberinto utilizando el modelo de *Physarum polycephalum*. [37]

Las reglas del modelo se describen en la Figura 22, mostrando los diferentes estados de los sitios y cómo interactúan durante las fases de desarrollo y búsqueda de alimento.

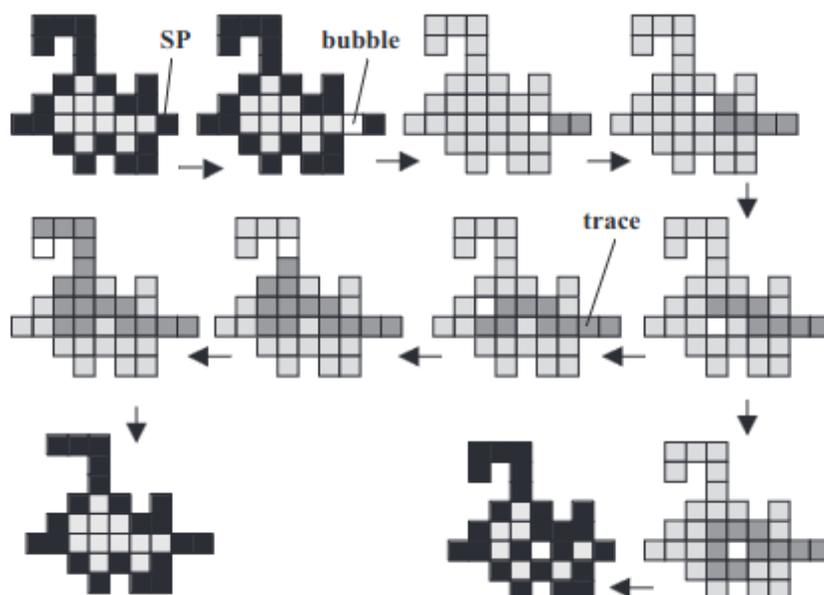


Figura 22: Reglas del modelo de célula mínima, mostrando los estados de los sitios y las interacciones. [37]

3.2. Robots para monitoreo poblacional

En esta sección, presentamos un resumen de los trabajos previos relacionados con robots o dispositivos aéreos para monitoreo poblacional, ya sea de animales o de humanos.

3.2.1. Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis

El artículo '*Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis*' proporciona una base robusta para contextualizar la tesis titulada "*Diseño de un autómata para monitoreo*" en el estado del arte. La revisión exhaustiva que presenta este artículo sobre la utilización de drones en diversas aplicaciones de monitoreo y manejo de recursos naturales en América Latina es directamente relevante para esta investigación, ya que ambos trabajos se enfocan en el desarrollo y aplicación de tecnologías autónomas para la recolección de datos ambientales [38].

En primer lugar, el artículo destaca cómo los drones, equipados con una variedad de sensores como Rojo, Verde y Azul (Red, Green, Blue, RGB), infrarrojos, multispectrales, hiperspectrales y LIDAR, han revolucionado la capacidad de los investigadores para monitorear con precisión diversos aspectos del medio ambiente. Estos avances permiten obtener datos de alta resolución espacial de manera rápida y a un costo reducido, lo cual es crucial para la efectividad y eficiencia del monitoreo ambiental. En el contexto de la tesis, el diseño de un autómata para monitoreo puede beneficiarse enormemente de estos conocimientos y tecnologías, aplicando principios similares de autonomía y precisión en la recolección de datos.

Además, el artículo proporciona una visión detallada de las ventajas y limitaciones de diferentes tipos de drones y sensores, ofreciendo información valiosa que puede influir en las decisiones de diseño y selección de componentes para el autómata. Por ejemplo, la comprensión de las capacidades y restricciones de los sensores hiperspectrales y LIDAR puede guiar la integración de tecnologías adecuadas en el sistema de monitoreo, asegurando una recopilación de datos eficiente y precisa. Esta información es crucial para el diseño de sistemas que requieran

alta resolución y precisión en la medición de variables ambientales.

3.2.2. Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields

El artículo '*Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields*' proporciona una base sólida para contextualizar la tesis titulada '*Diseño de un autómata para monitoreo*' en el estado del arte. La investigación presentada en este documento aborda el desarrollo y evaluación de varios algoritmos de muestreo para robots móviles, destinados a la detección de insectos en campos agrícolas, una problemática directamente relevante para la tesis, que se centra en el diseño de un autómata para el monitoreo ambiental. [39]

En primer lugar, el artículo destaca la importancia de los algoritmos de muestreo para maximizar la eficiencia en la detección de plagas, considerando las limitaciones de recursos como el tiempo y la energía. Los algoritmos desarrollados, tanto aquellos que operan sin información previa como los que utilizan datos en tiempo real, ofrecen estrategias para optimizar la recolección de datos en entornos agrícolas. En el contexto de la tesis, estos conocimientos pueden ser aplicados al diseño del autómata, integrando algoritmos de muestreo dinámico que prioricen puntos de muestreo estratégicos basados en patrones de distribución de plagas, mejorando así la eficiencia y precisión del monitoreo. [39]

Además, el artículo proporciona una evaluación detallada de la efectividad de estos algoritmos en diferentes escenarios de simulación, considerando variables como el tamaño del campo y la tasa de propagación de insectos. Esta información es crucial para la tesis, ya que ofrece una comprensión profunda de cómo los diferentes algoritmos pueden ser implementados y ajustados según las condiciones específicas del entorno de monitoreo. La integración de estas estrategias en el diseño del autómata permitirá una adaptación más rápida y precisa a las condiciones cambiantes del campo, asegurando una detección temprana y gestión eficaz de plagas.

El artículo también incluye estudios de caso basados en datos reales de infestaciones de insectos, proporcionando ejemplos prácticos y lecciones aprendidas que pueden ser directamente aplicables a la tesis. Estos estudios demuestran cómo la implementación de algoritmos de muestreo dinámico ha llevado a mejoras significativas en la eficiencia y precisión del monitoreo, validando la relevancia y aplicabilidad del diseño del autómata en contextos agrícolas reales. [39]

3.2.3. The Role of Robots in Environmental Monitoring

El artículo "The Role of Robots in Environmental Monitoring" por Robert Bogue, publicado en *Industrial Robot: An International Journal*, detalla la creciente utilización de sistemas robóticos en la monitorización ambiental. El artículo comienza con una introducción sobre la importancia creciente de los robots en este campo, seguida de un examen exhaustivo de varios tipos de sistemas robóticos utilizados para fines ambientales. Destaca los roles de los robots aéreos en la monitorización de la contaminación atmosférica y discute las capacidades de los robots de superficie y submarinos en la monitorización de ambientes acuáticos. Además, proporciona ejemplos de aplicaciones de monitorización robótica terrestre. El documento concluye resumiendo los avances y contribuciones significativas de los robots en la provisión de una cobertura de datos espaciales y temporales mejorada, la detección de contaminación, la caracterización de condiciones ambientales y la localización de actividades ilícitas.

Los robots han mejorado significativamente los métodos tradicionales de monitorización ambiental, ofreciendo datos con mayor precisión y cobertura. El uso de drones equipados con dispositivos de imagen y sensores pequeños y ligeros ha revolucionado la detección de contaminantes en el aire y la caracterización de entornos acuáticos y terrestres. Además, la integración de técnicas de IA ha mejorado la eficiencia y efectividad de las imágenes de drones ambientales. El artículo también enfatiza la importancia de los robots acuáticos en la monitorización de entornos de agua dulce y marinos, desde despliegues locales a corto plazo hasta misiones oceánicas de larga duración. En general, el artículo ofrece una visión general exhaustiva de las diversas y crecientes aplicaciones de los robots en la monitorización ambiental, subrayando su papel crítico en la ciencia ambiental moderna. [40]

4. Propuesta a desarrollar

El proyecto propuesto consiste en el desarrollo de un sistema de monitoreo poblacional basado en la implementación de un autómata en un modelo bidimensional no lineal. En este caso, como mencionamos anteriormente, el algoritmo está basado en el modelo de Physarum Polycephalum. Este modelo es un organismo unicelular que se comporta como un autómata celular, y es capaz de resolver problemas de optimización y ruteo.

A su vez, el sistema propuesto se basa en la utilización de robots autónomos, los cuales se encargarán de recolectar información de la población y de los entornos en los que se encuentran. Estos robots estarán equipados con cámaras y sensores que les permitirán detectar y clasificar entidades poblacionales. Además, los robots estarán conectados a una red de comunicación que les permitirá compartir información en tiempo real.

El sistema funcionará de la siguiente manera: los robots autónomos recibirán la ruta a seguir por parte de nuestro simulador del Physarum Polycephalum, el cual se encargará de determinar la ruta óptima para recolectar información de la población. Una vez que los robots recolecten la información, esta será enviada a un servidor central, el cual se encargará de procesar la información y de generar reportes en tiempo real.

La implementación de este sistema permitirá a los investigadores y a las autoridades locales monitorear poblaciones de manera eficiente y en tiempo real. Además, el sistema permitirá la detección de cambios en las poblaciones y en los entornos en los que se encuentran.

Por ello tendremos principalmente dos 'productos' a desarrollar, el primero será el simulador del Physarum Polycephalum, el cual será un sistema que permitirá determinar rutas óptimas para recolectar información de la población. El segundo producto será el sistema de monitoreo poblacional, el cual será un sistema que permitirá a los robots autónomos recolectar información de la población y de los entornos en los que se encuentran.

Se detallará la implementación de estos sistemas en las siguientes secciones.

4.1. Requerimientos

Para el desarrollo de nuestro Trabajo Terminal (TT), es necesario establecer los requerimientos que debe cumplir el robot que simulará el comportamiento del Physarum Polycephalum. En las siguientes secciones, se describirán los requerimientos funcionales y no funcionales que se deben cumplir en nuestro Trabajo Terminal (TT).

4.1.1. Requerimientos funcionales

En esta sección se presentan los requerimientos funcionales que definen las características y capacidades específicas que el sistema debe proporcionar para cumplir su propósito en el contexto de simulación y monitoreo de rutas automatizadas. Estos requerimientos aseguran que el sistema cumpla con las funciones clave necesarias para la interacción y operación de la simulación. Los requerimientos funcionales se listan en el Cuadro 3.

ID	Nombre	Requerimiento Funcional
RF1	Seleccionar	Permitir al usuario seleccionar estados en el simulador mediante el teclado y el ratón.
RF2	Colocar	Posibilitar la colocación de los estados inicial y final en el lienzo antes de iniciar la simulación.
RF3	Iniciar	Habilitar la simulación de rutas al presionar la tecla ENTER.
RF4	Cargar	Proveer la opción de cargar un mapa o imagen en el lienzo para definir el entorno inicial de la simulación.
RF5	Visualizar	Mostrar la ruta generada en tiempo real durante la simulación.

Cuadro 3: Requerimientos funcionales del sistema

4.1.2. Requerimientos no funcionales

En esta sección se describen los requerimientos no funcionales, los cuales establecen los criterios de calidad y desempeño que el sistema debe cumplir para garantizar una operación robusta, eficiente y compatible en distintos entornos. Estos requerimientos no funcionales aseguran que el sistema sea eficiente, adaptable y compatible, proporcionando una experiencia de usuario satisfactoria. Los requerimientos no funcionales se detallan en el Cuadro 4.

ID	Nombre	Requerimiento No Funcional
RNF1	Rendimiento	El sistema debe responder a las solicitudes de inicio de simulación en menos de 3 segundos para mapas de hasta 1000 nodos.
RNF2	Escalabilidad	El sistema debe poder manejar simulaciones en mapas con hasta 5000 nodos sin una caída de rendimiento superior al 5 %.
RNF3	Portabilidad	El sistema debe ser portable y ejecutarse correctamente en sistemas operativos Windows 10 y Linux basados en Debian.
RNF4	Facilidad de uso	El sistema debe ser comprensible y usable para un usuario novato tras un máximo de 30 minutos de uso guiado.

Cuadro 4: Requerimientos No funcionales del sistema

4.2. Diagramas

En esta sección se presentan todos los diagramas necesarios para la implementación del sistema propuesto. En primer lugar, se presenta un diagrama de arquitectura del sistema propuesto, el cual muestra la interacción entre los diferentes componentes del sistema.

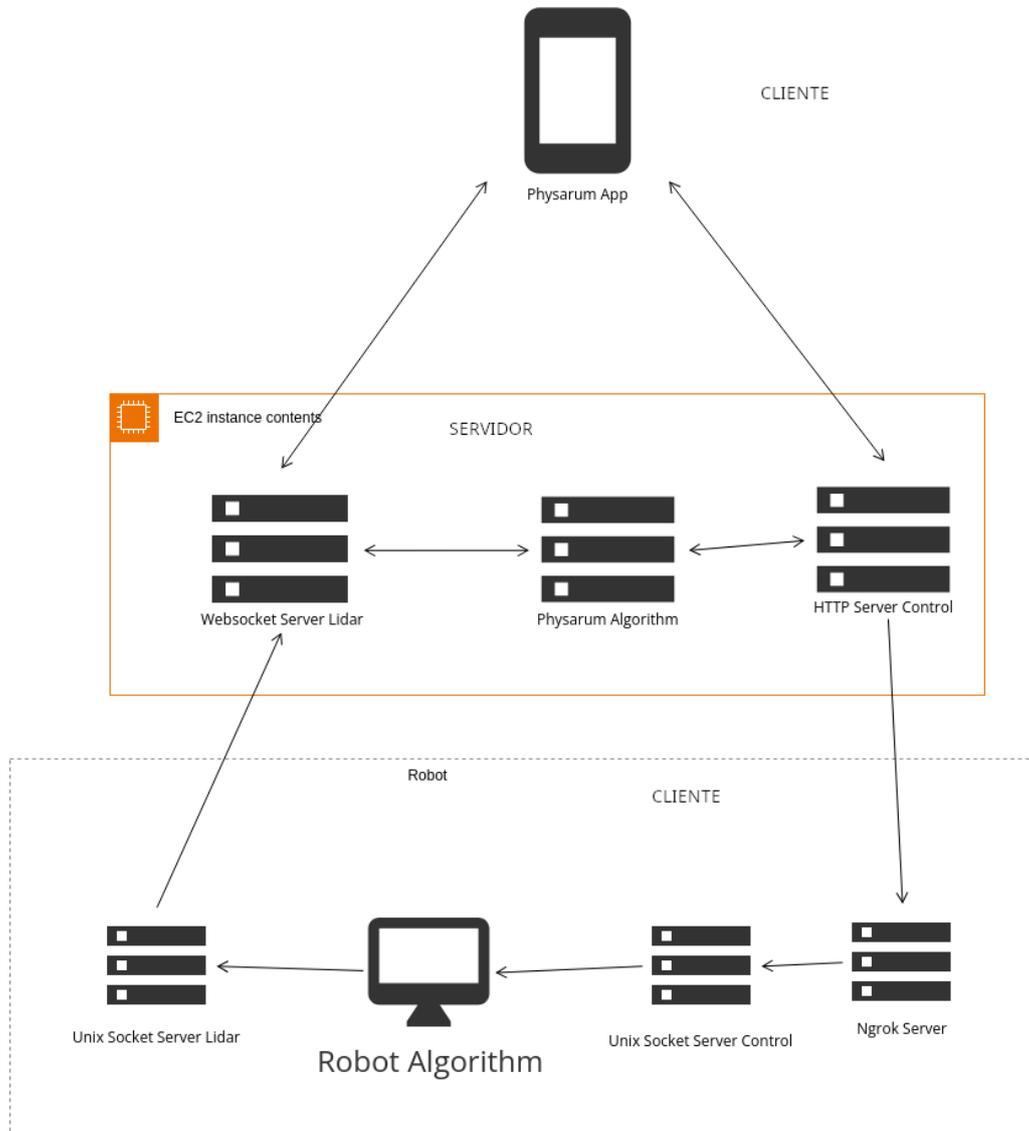


Figura 23: Diagrama de arquitectura del sistema propuesto.

Como se muestra en la Figura 23, el sistema mostrado en el diagrama representa una **Arquitectura Distribuida Cliente-Servidor Multinivels** implementada para el control remoto de un robot mediante un algoritmo basado en **Physarum polycephalum** y la utilización de datos obtenidos por un sensor LiDAR. La arquitectura se compone de tres capas principales: la capa de la aplicación móvil, la capa de procesamiento en la nube y la capa física del robot.

La **Physarum App** es la interfaz de usuario a través de la cual se envían los comandos al sistema, los cuales son gestionados por un servidor Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP) ubicado en una instancia EC2 de Amazon Web Services. Esta instancia contiene tres servidores: un servidor HTTP que recibe los comandos de control, un servidor WebSocket que maneja la recepción de los datos del sensor LiDAR enviados des-

de el robot, y el núcleo del sistema, el algoritmo de **Physarum**, encargado de procesar esta información para la toma de decisiones en tiempo real sobre el comportamiento del robot.

En la capa física, el robot está controlado mediante dos servidores Unix Socket. El **Unix Socket Server LiDAR** recibe y transmite los datos del sensor LiDAR al servidor WebSocket en la instancia EC2, mientras que el **Unix Socket Server Control** se encarga de recibir los comandos procesados y enviarlos al robot. Además, se utiliza un servidor Ngrok que permite la conexión remota segura, facilitando el control del robot desde la aplicación móvil.

Esta arquitectura distribuida permite la integración fluida de los componentes del sistema, asegurando una correcta interacción entre los datos sensoriales, el procesamiento en la nube y el control remoto del robot.

Después de la presentación de la arquitectura del sistema, se presentan los diagramas de flujo de las Figuras 24 - 26.

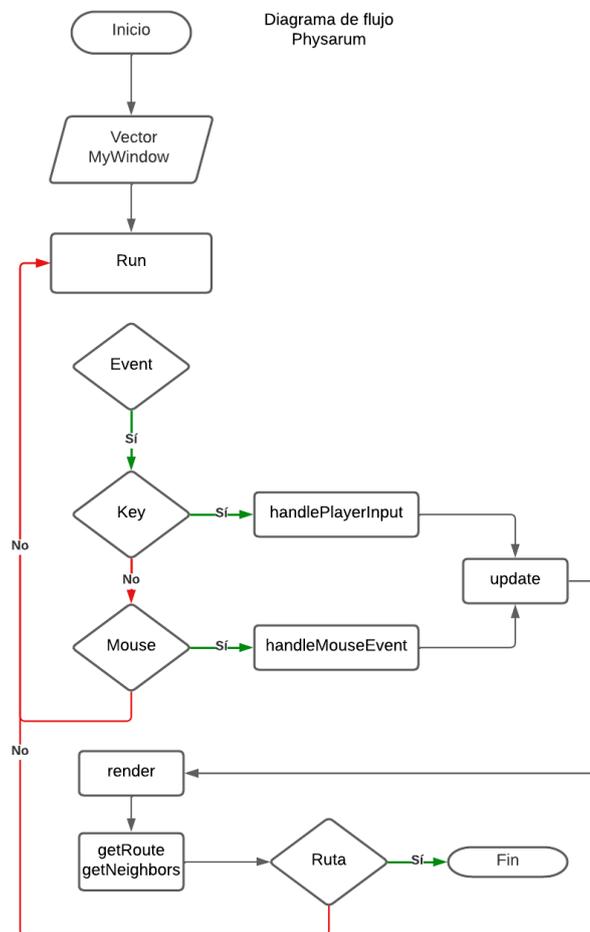


Figura 24: Diagrama de flujo del simulador de **Physarum polycephalum**.

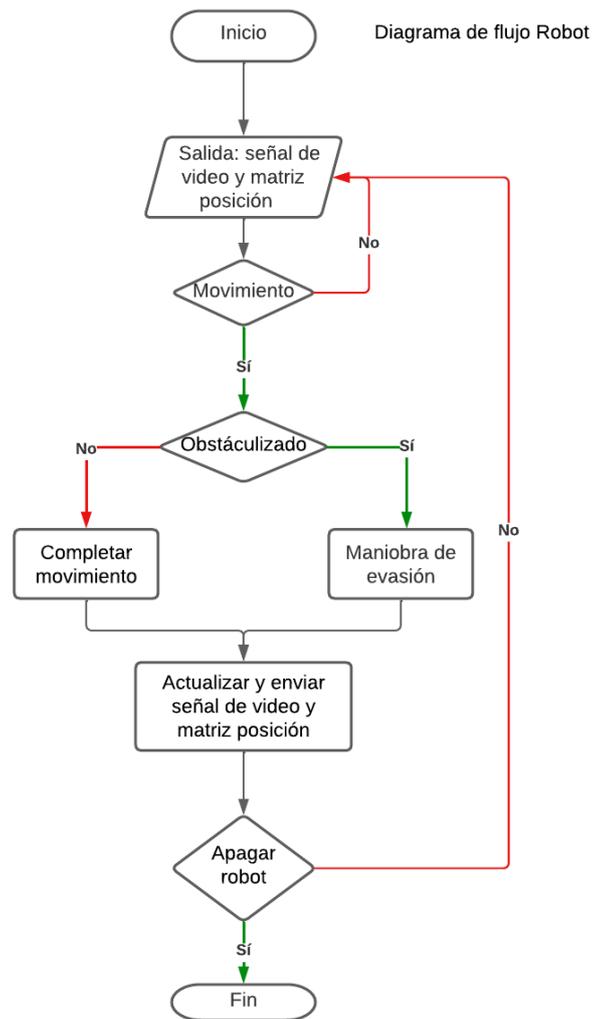


Figura 25: Diagrama de flujo del sistema del robot autónomo.

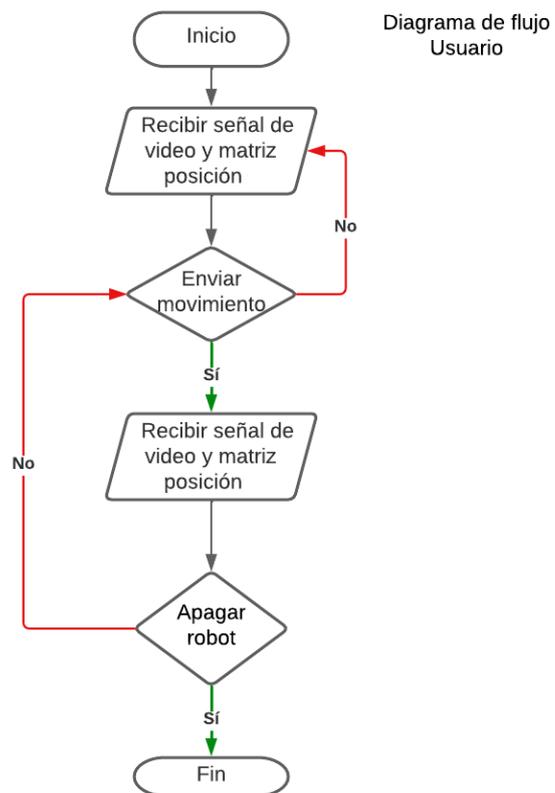


Figura 26: Diagrama de flujo de la aplicación móvil.

Después de la presentación de los diagramas de flujo, en la Figura 27 se presenta el diagrama de clases del sistema propuesto.

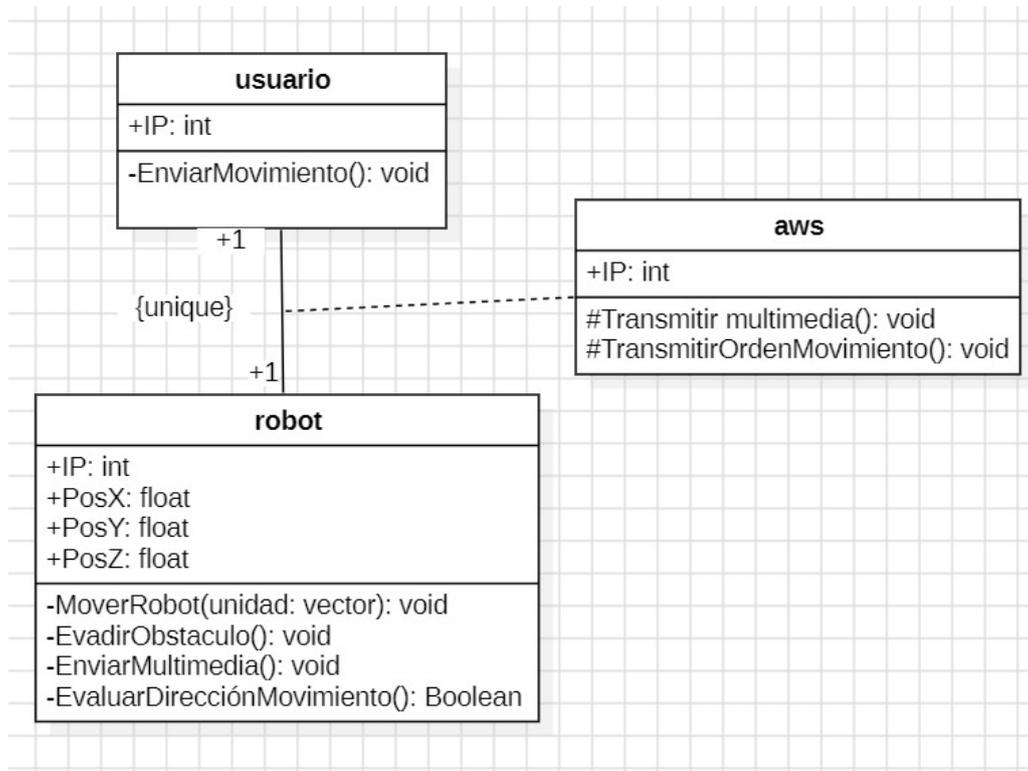


Figura 27: Diagrama de clases del sistema propuesto.

El diagrama de clases muestra la interacción entre el usuario, el robot y un servidor en Amazon Web Services (AWS). El **usuario** tiene un atributo para su dirección IP y un método *EnviarMovimiento()* que le permite enviar órdenes al **robot**. Este último, con atributos para su IP y su posición en los ejes X, Y y Z, tiene métodos para moverse (*MoverRobot()*), evitar obstáculos (*EvadirObstaculo()*), transmitir multimedia (*EnviarMultimedia()*) y evaluar su dirección de movimiento (*EvaluarDirecciónMovimiento()*).

El **servidor en AWS** cuenta con métodos para transmitir multimedia y órdenes de movimiento hacia el robot. El servidor actúa como intermediario entre el robot y el sistema, facilitando la transmisión de datos y el control remoto de manera segura y eficiente. El sistema asegura una comunicación única y directa entre cada usuario y su robot, manteniendo una interacción fluida y controlada.

En la Figura 28 se presenta el diagrama de casos de usos del sistema propuesto.

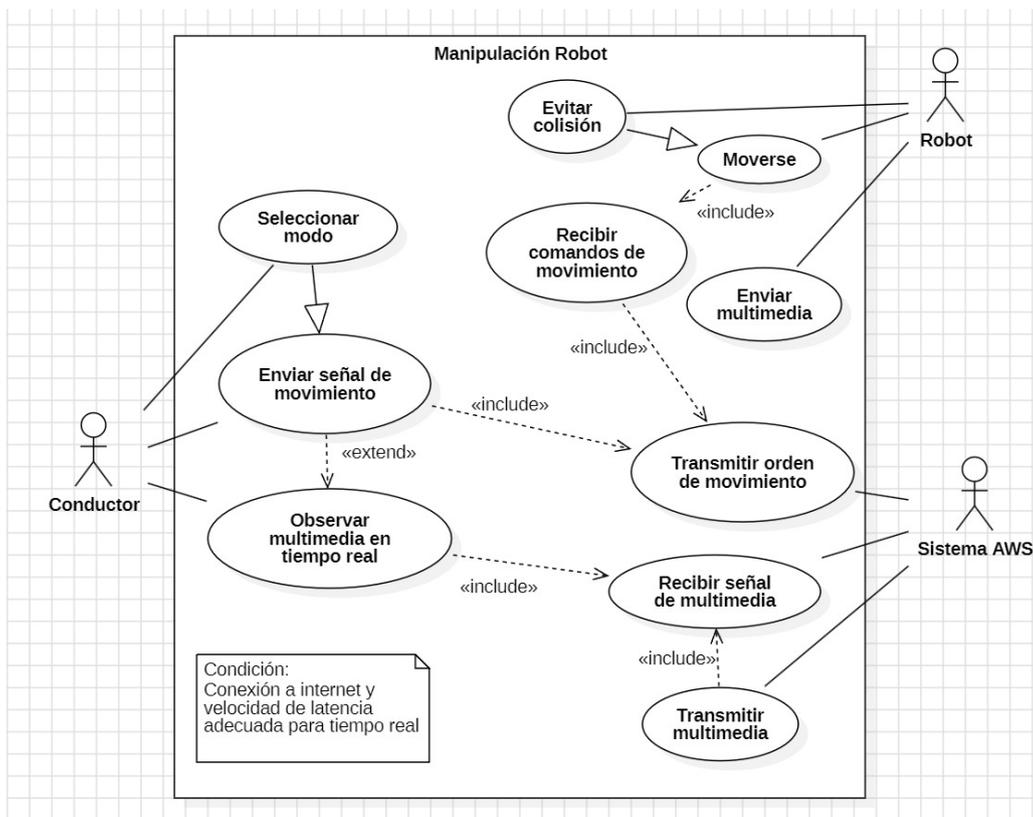


Figura 28: Diagrama de casos de uso del sistema propuesto.

Y el Cuadro 5 muestra la descripción del caso de uso **Manipulación del Robot**.

Cuadro 5: Descripción del Caso de Uso: Manipulación del Robot

Identificador	CU-01 Manipulación del Robot
Descripción	El conductor controla los movimientos del robot, seleccionando el modo de operación, enviando comandos de movimiento y observando la transmisión multimedia en tiempo real mientras AWS transmite las órdenes y multimedia.
Actores	Conductor, Robot, Sistema AWS
Precondiciones	<ul style="list-style-type: none"> ■ El sistema debe estar conectado a internet. ■ La latencia de la red debe ser adecuada para la transmisión en tiempo real.
Postcondiciones	<ul style="list-style-type: none"> ■ El robot ejecuta las órdenes de movimiento enviadas por el conductor. ■ El conductor puede observar la transmisión multimedia en tiempo real.
Secuencia Normal	<p>A) El conductor selecciona el modo de operación del robot.</p> <p>B) El conductor envía una señal de movimiento.</p> <p>C) El sistema AWS recibe la señal y la transmite al robot.</p> <p>D) El robot recibe la señal y ejecuta el movimiento según las instrucciones.</p> <p>E) El robot evita colisiones mientras se mueve.</p> <p>F) El robot transmite la señal multimedia en tiempo real.</p> <p>G) El conductor observa la multimedia transmitida en tiempo real.</p>

Excepciones	<p>A) Si la conexión a internet se pierde o es inestable, el sistema notifica al conductor sobre la interrupción en la transmisión.</p> <p>B) Si el robot detecta un obstáculo ineludible, detiene su movimiento y espera nuevas instrucciones.</p>
Rendimiento	<ul style="list-style-type: none"> ■ El sistema debe procesar las órdenes de movimiento en menos de 1 segundo. ■ La transmisión de la señal multimedia debe tener un máximo de 300ms de latencia.
Frecuencia	Se espera que este caso de uso se realice continuamente durante la operación del robot.
Importancia	Vital
Urgencia	Inmediata, ya que el sistema debe responder en tiempo real.
Comentarios	El sistema depende de la calidad de la conexión a internet para mantener la comunicación en tiempo real entre el conductor y el robot.

Finalmente, en la Figura 29 se presenta el diagrama de secuencia del sistema propuesto.

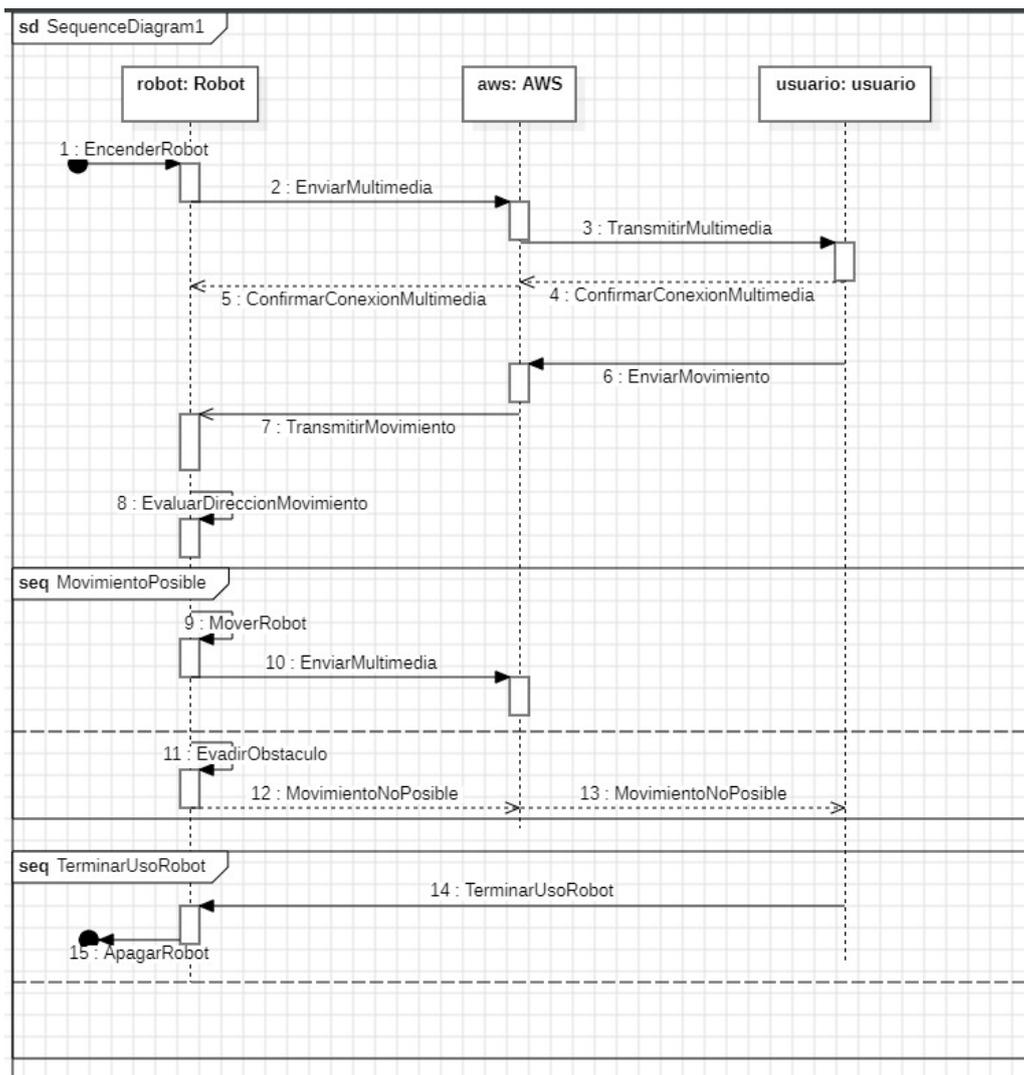


Figura 29: Diagrama de secuencia del sistema propuesto.

El diagrama de secuencia muestra la interacción entre el **robot**, el **servidor AWS**, y el **usuario**. Comienza cuando el usuario enciende el robot, lo que desencadena la transmisión de multimedia desde el robot hacia AWS, que luego retransmite al usuario. Tras esto, se confirma la conexión de multimedia entre el robot y AWS, asegurando que el sistema esté listo para recibir comandos.

El usuario puede entonces enviar órdenes de movimiento al robot a través de AWS. El robot evalúa la dirección del movimiento, y si es posible, procede a moverse mientras sigue enviando multimedia. En caso de encontrar un obstáculo, el robot intenta evadirlo. Si el movimiento no es posible, el robot notifica al usuario sobre la situación.

Finalmente, cuando el usuario desea terminar la sesión, se envía la señal para cerrar el uso del robot, lo que lleva al apagado del mismo. Así, el diagrama cubre desde el encendido y transmisión de multimedia hasta la ejecución de movimientos y finalización del uso del robot.

5. Implementación

Como mencionamos en el Marco Teórico, nos centraremos en el simulador del Physarum Polycephalum y en el robot con una Raspberry Pi. En este caso, el simulador del Physarum Polycephalum se encargará de determinar la ruta óptima para recolectar información de la población. Por otro lado, el robot con una Raspberry Pi se encargará de recolectar información de la población y de los entornos en los que se encuentran.

En este capítulo, se detallará la implementación de estos sistemas. Primero, se describirá la implementación del simulador del Physarum Polycephalum. Luego, se describirá la implementación del robot con una Raspberry Pi. Finalmente, se describirá la integración de estos sistemas.

5.1. Simulador del Physarum Polycephalum

El algoritmo propuesto en este documento está inspirado en el modelo de agentes de Jones [36]. El algoritmo es un autómata celular que simula el comportamiento de *Physarum polycephalum* en un laberinto, por lo que necesitamos definir algunos conceptos. Sea \mathbb{Z} el conjunto de los números enteros, y definamos la longitud de una tupla x como $|x|$. Para todas las tuplas x y y donde $|x| = |y|$, denotamos $x \oplus y$ como el resultado de la suma de cada componente de x y y , es decir, $(x \oplus y)_i = x_i + y_i$ para todo $i \in \mathbb{Z}$.

Un autómata celular se define como una tupla (\mathbb{Z}^n, S, N, f) donde n es la dimensión tal que $n \in \mathbb{Z}^+$, S es un conjunto de estados finito y no vacío, N es un conjunto no vacío y finito de vecindarios pertenecientes a \mathbb{Z}^n , y f es una función de transición local, es decir, $f : S^N \rightarrow S$ donde S^N representa el conjunto de todas las configuraciones posibles de estados de vecindarios en N .

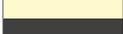
Así, el algoritmo propuesto en este trabajo se define como un autómata celular (\mathbb{Z}^2, S, N, f) donde $n = 2$, $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}^9$, y $f : \{0, 1, 2, 3, 4, 5, 6, 7, 8\}^9 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

Sea $P = (C(x, y : t), N(x, y : t), M(x, y : t))$ el estado combinado de la célula central, su vecindario y su memoria en la posición (x, y) en el tiempo t . Aquí, $C(x, y : t)$ representa el estado actual de la célula central, $N(x, y : t)$ es el conjunto de estados de las celdas vecinas adyacentes, y $M(x, y : t)$ es el estado de memoria asociado a la célula.

La función de transición f toma como entrada P y actualiza el estado de la célula central en la siguiente generación de acuerdo con las siguientes reglas:

$$f(P) = \begin{cases} 7 & \text{si } C(x, y : t) = 0 \wedge \exists s \in N(x, y : t) \text{ tal que } s \in \{3, 4, 6\} \wedge M(x, y : t) = 0, \\ 6 & \text{si } C(x, y : t) = 1 \wedge \exists s \in N(x, y : t) \text{ tal que } s \in \{5, 6\}, \\ 5 & \text{si } C(x, y : t) = 4 \wedge \exists s \in N(x, y : t) \text{ tal que } s \in \{3, 5, 6\} \\ & \wedge M(x, y : t) = 0 \wedge \forall s \in N(x, y : t), s \notin \{0, 7\}, \\ 0 & \text{si } C(x, y : t) = 5 \wedge M(x, y : t) \notin \{5, 8\} \wedge \forall s \in N(x, y : t), s \notin \{1, 3, 4, 6\}, \\ 8 & \text{si } C(x, y : t) = 5 \wedge \text{no se cumple la condición anterior,} \\ 4 & \text{si } C(x, y : t) = 7 \wedge \exists s \in N(x, y : t) \text{ tal que } s \in \{3, 4, 6\}, \\ 5 & \text{si } C(x, y : t) = 8, \\ C(x, y, t) & \text{en otro caso.} \end{cases}$$

Donde los estados del autómata celular se definen en el **Cuadro 6**.

Color	Estado	Descripción
	0	Campo libre
	1	Nutriente no encontrado
	2	Repelente
	3	Punto inicial
	4	Gel contrayéndose
	5	Gel compuesto
	6	Nutriente encontrado
	7	Expansión de Physarum
	8	Gel no compuesto

Cuadro 6: Estados del autómata celular

En la fuente mencionada, se detalla el algoritmo básico de Physarum Polycephalum, diseñado originalmente para con sistemas multiagentes. Sin embargo, en la versión propuesta aquí, optamos por realizar un autómata celular de 2 dimensiones con la vecindad de Moore, facilitando así el acceso a un mayor número de vecinos para comparación y permitiendo obtener una perspectiva más clara de la dirección óptima para el desplazamiento del agente.

Sin embargo, el uso del vecindario de Moore en lugar del vecindario de von Neumann introduce ciertos desafíos no presentes en el algoritmo original. Uno de estos desafíos surge en las esquinas (NW, NE, SW, SE), donde el repelente podría permitir que el agente escape, contrario a lo deseado. Para abordar este inconveniente, se implementó una solución que consiste en colocar un repelente imaginario en la esquina cuando dos esquinas adyacentes presentan repelentes en un ángulo de 90° entre sí. Este ajuste permite la creación de una gama más amplia de formas, como se ilustra en las Figuras 30, 31 y 32. En estas imágenes, el número total de células es de 50 x 50.

Gracias a la resolución de la fuga en las esquinas por nuestro algoritmo, es posible generar mapeos más diversos de cuevas y catacumbas. Este enfoque mejora significativamente nuestra comprensión de la topografía del área explorada. Además, la diversidad en el mapeo facilita la identificación del número y variedad de caminos disponibles, lo que se ha implementado mediante un algoritmo de mapeo de imágenes que ayuda en la representación gráfica de dicha topografía, como se muestra en la Figura 33.

También el algoritmo ha sido probado en un entorno real, donde ha sido capaz de generar rutas óptimas en la Catacumba de París, como se muestra en la Figura 34. El espacio explorado por el algoritmo es de 1000 x 1000 células.

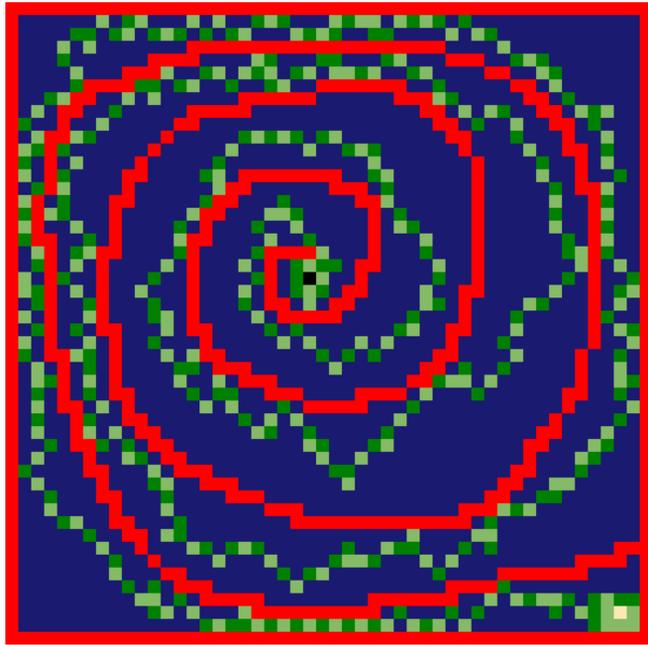


Figura 30: Physarum Polycephalum resolviendo un laberinto en espiral.

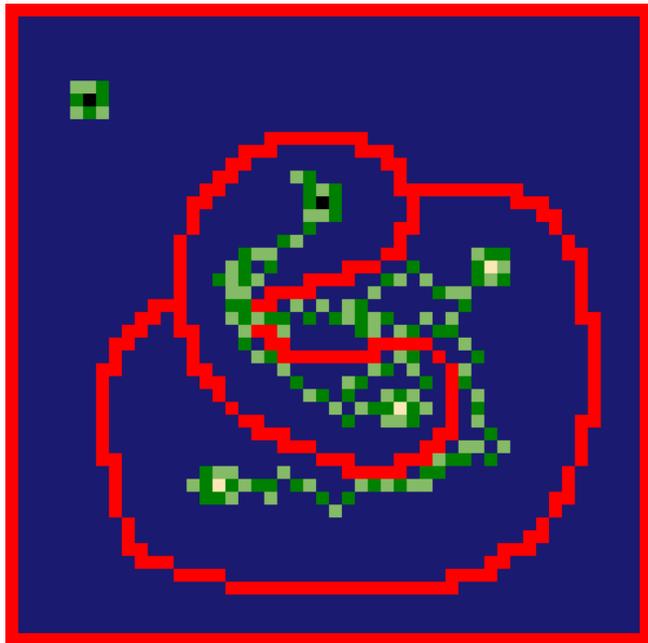


Figura 31: Physarum Polycephalum resolviendo un laberinto de tipo circular.

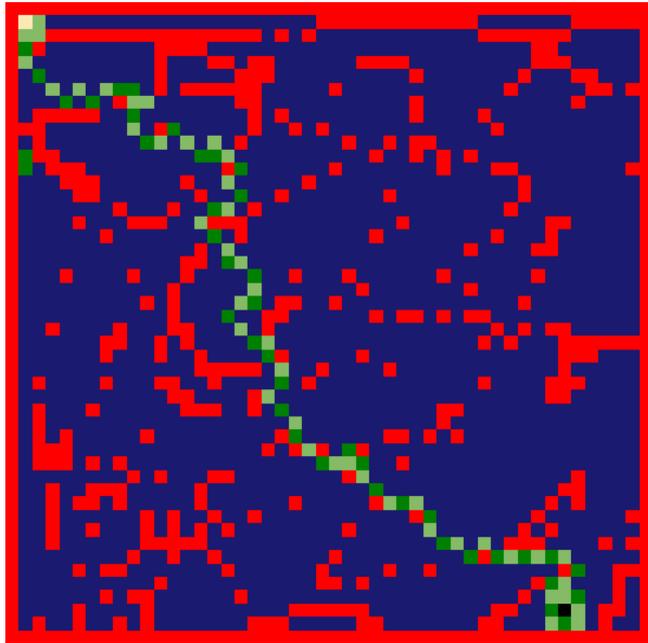


Figura 32: Physarum Polycephalum resolviendo un laberinto con obstáculos.

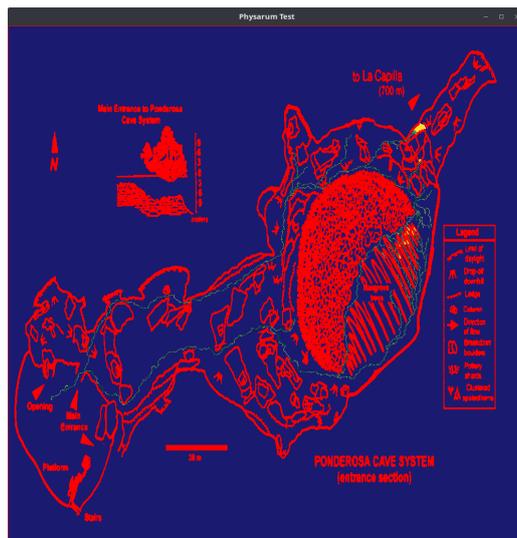


Figura 33: Mapeo del sistema de cuevas usando Physarum Polycephalum.

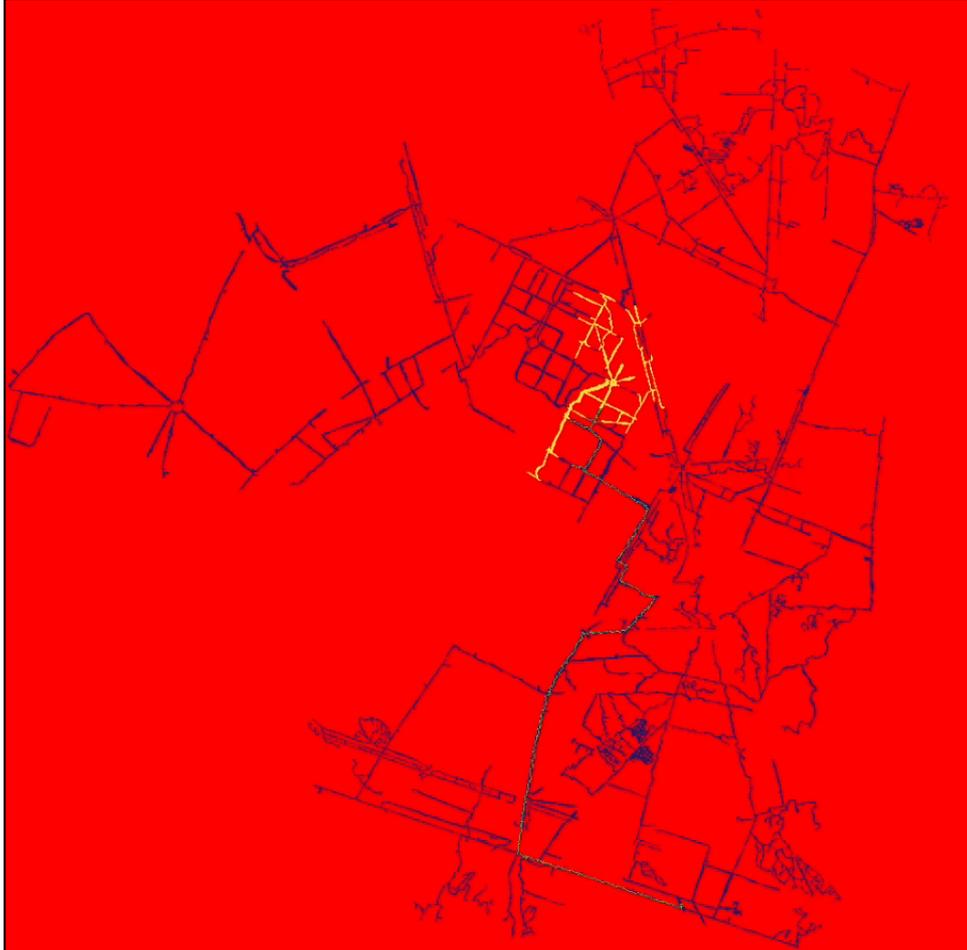
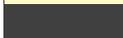
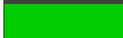
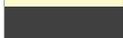


Figura 34: Mapeo de la catacumba usando *Physarum Polycephalum*, utiliza 5264 pasos para obtener la ruta.

Como se puede observar en los Cuadros 7 y 8, el algoritmo propuesto es capaz de resolver laberintos de manera eficiente, generando rutas óptimas en entornos complejos y desconocidos. Además, el algoritmo es capaz de adaptarse a diferentes topografías, lo que lo convierte en una herramienta versátil para la exploración de entornos desconocidos y nos es de ayuda en el monitoreo de poblaciones y sistemas relacionados.

Color	State	Initial State Catacomb
	0	32,891
	1	1
	2	967,105
	3	2
	4	0
	5	0
	6	0
	7	0
	8	0

Cuadro 7: Estado inicial de la catacumba

Color	State	Initial State Cave
	0	853,861
	1	3
	2	146,135
	3	1
	4	0
	5	0
	6	0
	7	0
	8	0

Cuadro 8: Estado inicial de la cueva

Cabe señalar que dado que el algoritmo está bioinspirado, la función que simula el comportamiento del plasmodio se asigna de manera pseudoaleatoria a un vecino adyacente, con una probabilidad de $1/8$. Esta característica permite que la expansión del algoritmo tome una forma circular en lugar de una expansión cuadrada o lineal. Sin embargo, al modificar la función de probabilidad, es posible lograr una expansión más irregular en lugar de simplemente circular.

En cuanto la implementación del algoritmo, se ha utilizado el lenguaje de programación C++ y la librería OpenCV para la obtención de imágenes. La parte de las esquinas que mencionamos con anterioridad se implementó como se muestra en el Listing 1:

Listing 1: Implementación del problema de las esquinas

```
1      std::vector<int> Physarum::isOnCorner(std::vector<int>
      neighboursData) {
2      std::vector<int> corners;
3      if (neighboursData[0] == 2 && neighboursData[2] == 2) {
4          corners.push_back(1);
5      }
6      if (neighboursData[0] == 2 && neighboursData[6] == 2) {
7          corners.push_back(7);
8      }
9      if (neighboursData[6] == 2 && neighboursData[4] == 2) {
10         corners.push_back(5);
11     }
12     if (neighboursData[2] == 2 && neighboursData[4] == 2) {
13         corners.push_back(3);
14     }
15     return corners;
16 }
```

En el Listing 1, se muestra la implementación de la función que detecta si el agente se encuentra en una esquina. La función recibe un vector de enteros que representa los estados de los vecinos adyacentes. Si dos esquinas adyacentes presentan repelentes, la función devuelve un vector con las esquinas en las que se encuentra el agente. En caso contrario, la función devuelve un vector vacío.

Por ello podemos decir que el algoritmo propuesto es capaz de resolver laberintos de manera eficiente, generando rutas óptimas en entornos complejos y desconocidos. Además, el algoritmo es capaz de adaptarse a diferentes topografías, lo que lo convierte en una herramienta versátil para la exploración de entornos desconocidos y nos es de ayuda en el monitoreo de poblaciones y sistemas relacionados.

5.1.1. Generación de rutas de nuestro simulador

El robot de monitoreo deberá de seguir una ruta preestablecida, la cual es calculada por el algoritmo del Physarum Polycephalum en el simulador que ha sido desarrollado.

Primeramente, se tiene un lienzo, el cual es representado por un arreglo de celdas de tamaño $n \times n$, donde n es el tamaño deseado para la representación del espacio en el cual el Physarum calculará la ruta una vez terminada la simulación de este.

En el lienzo, se colocan los estados sobre el lienzo por medio del teclado y el mouse, donde en el teclado son presionadas las teclas de 1 al 9 para poder elegir cada uno de los estados que puede tomar la celda en la cual se haya presionado el botón izquierdo del mouse. Los estados que tienen mayor relevancia y que son los que se deben de colocar para poder realizar la simulación correctamente son los 1 y 4, debido a que representan el nutriente como el punto inicial respectivamente. El punto inicial es de donde se empezará con la expansión del Physarum, mientras que el nutriente es el destino final, puesto que una vez encontrado, cambiara su estado al estado 6, el cual es correspondiente al estado de nutriente encontrado. Y una vez finalizada la simulación, el algoritmo se detendrá automáticamente y quedará plasmada la ruta por la cual

el Physarum encontró el o los nutrientes desde el punto inicial, la cual será enviada al robot para su posterior seguimiento para llegar a su destino en el mundo real.

Al iniciar el programa, se despliega una pantalla la cual muestra el lienzo con el espacio que hemos predefinido anteriormente en el código. En este espacio al inicio se pueden colocar los diferentes estados como se mencionó anteriormente, por lo que una vez se haya colocado la configuración deseada, para iniciar la simulación se presiona en el teclado la tecla ENTER. El algoritmo se empieza a expandir, siendo aplicadas cada una de las reglas en cada una de las celdas del arreglo. El algoritmo termina cuando la ruta es encontrada y el Physarum termina de contraerse, dicha ruta es la que es enviada al robot para su seguimiento. Primeramente, se iniciaron las simulaciones en espacios pequeños. La primera configuración como se puede ver en la Figura 35 fue la de un lienzo de tamaño 10 x 10.

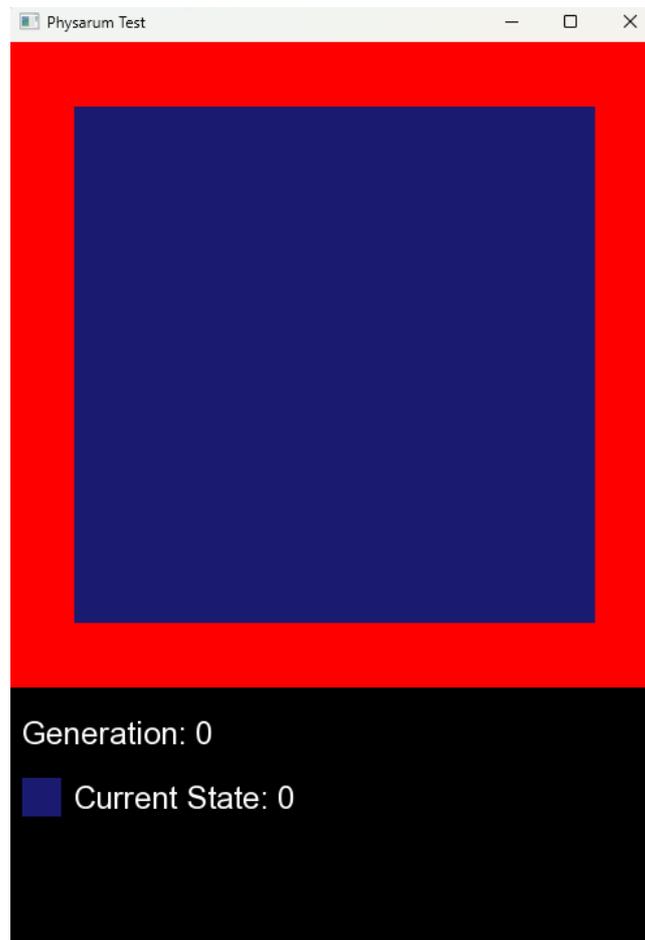


Figura 35: Lienzo de 10 x 10 celdas '1'

Posteriormente se colocaron los estados correspondientes a el estado inicial y al nutriente no encontrado. Como se puede ver en la Figura 36.

Una vez colocada la configuración inicial, entonces se procede a iniciar la simulación, con la cual, al terminar las iteraciones se genera una ruta que va desde el estado inicial al nutriente no encontrado, el cual para este momento ha cambiado su estado a nutriente encontrado, cambiando a su vez el color correspondiente a este estado. Como se puede ver en la Figura 37.

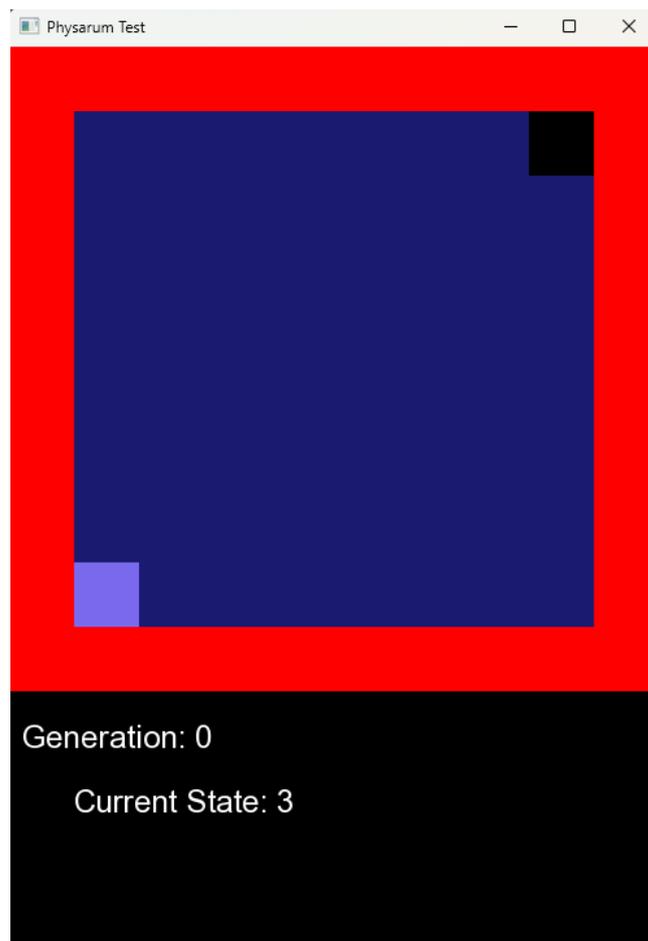


Figura 36: Estados iniciales de la simulación '1'



Figura 37: Ruta encontrada en la simulación '1'

En el programa es posible ver en qué generación es en la que se encuentra el algoritmo, además del estado en el cual en ese momento se tiene elegido en el teclado, esto para ser colocado en el lienzo.

Es posible colocar otra configuración una vez reiniciando el programa, donde se puede colocar los estados en distinta posición, por lo que se muestran a continuación distintas configuraciones una vez finalizadas las simulaciones correspondientes a cada una de ellas. Esto se puede ver en las Figuras 38 - 40.



Figura 38: Resultados de una segunda configuración

Las simulaciones realizadas nos han devuelto resultados satisfactorios, debido a que las rutas son correctamente generadas de acuerdo a las reglas plasmadas en el algoritmo, yendo del punto inicial al final, y finalizando una vez que el Physarum ya no tiene la necesidad de expandirse, debido a que ha encontrado el nutriente que estaba buscando.



Figura 39: Resultados de una tercera configuración



Figura 40: Resultados de una cuarta configuración

5.1.2. Codificación e implementación de algoritmo en el robot en la primera iteración 1

Con las rutas que han sido generadas de acuerdo a cada una de las simulaciones que fueron ejecutadas en el programa, como ya se ha mencionado, se genera la información con la cual el robot realizará el seguimiento de esta ruta para llegar de un punto inicial al final. Esta información corresponde a la ruta que genera el Physarum cuando termina la simulación de llegar desde su punto inicial de expansión a un nutriente con el cual se alimenta.

Para poder hacer una correcta manipulación de la información, la cual pueda ser enviada al robot que este pueda interpretarla y avanzar de acuerdo la ruta generada, es necesario crear un espacio el cual represente la posición del robot en un determinado lugar, junto con el destino al cual se quiere llegar, a partir de ahí pasar esta configuración al modelo del Physarum, ejecutar la simulación y obtener la ruta, finalizando con la recopilación de la información relacionada con la ruta y su almacenamiento, lo cual es representado en el Listing 2.

Listing 2: Pseudo Código de las rutas

```
1   Iniciar programa
2   Colocar la configuraci'on inicial
3   Coloca punto inicial
4   Coloca punto final
5   Ejecutar Simulador_Physarum
6   Si Simulador_Physarum obtuvo ruta, entonces:
7       Guardar las coordenadas del punto inicial
8       Guardar las coordenadas del punto final
9       Guardar las coordenadas de las c'elulas por orden de
        aparici'on
10      Desplegar todas las coordenadas en un archivo
11  Si no
12      Finaliza programa
13  Fin Si
```

A partir de lo anterior, se comprende que al robot se le será enviado las coordenadas correspondientes a la ruta obtenida por el simulador del Physarum, con un punto inicial, el cual marcará su posición actual, su punto final, representando el destino al cual llegará el robot y las células del Physarum ordenadas por orden de aparición, las cuales son las coordenadas por las cuales el robot tendrá que pasar para llegar desde el punto inicial al final.

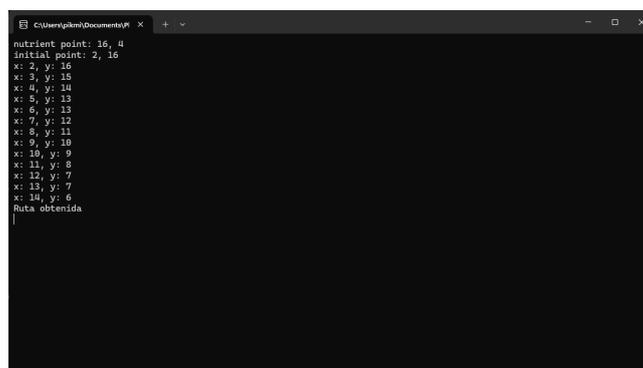
Con lo anterior, es posible la creación de información la cual el robot sea capaz de leer y a partir de sus propias funciones, llegar de un punto inicial al final. Esta información es almacenada en un archivo, el cual contiene cada una de las coordenadas ordenadas, siendo la primera el punto inicial, posteriormente cada una de las demás coordenadas que representan a cada una de las células del Physarum, ordenadas por orden de aparición para que el robot tenga un orden el cual seguir para poder llegar a su destino.

Finalmente, la última coordenada corresponde a la coordenada del punto final, el cual es el destino al cual el robot deberá de llegar, finalizando así su recorrido.

5.1.3. Codificación e implementación de algoritmo en el robot en la primera iteración 2

La implementación en el robot no cambia respecto a cómo se estuvo planeando desde el inicio, debido a que las rutas son obtenidas de acuerdo con la simulación del Physarum, que, una vez finalizada esta, a partir de una serie de condiciones y convirtiendo el camino en coordenadas las cuales el robot pueda interpretar para realizar su recorrido de la mejor forma posible, se pueda llegar de un punto inicial a un punto final de forma correcta.

Como se planteó desde el inicio, el algoritmo debe generar su ruta con ayuda del Physarum, y a partir de esta, generar una serie de coordenadas las cuales el robot pueda seguir para moverse de un punto al otro. En la implementación, el archivo generado a partir de la ruta que se obtuvo con el Physarum, se puede ver en la Figura 41.



```
C:\Users\pkm\Documents\... x + - - □ x
nutrient point: 16, 4
initial point: 2, 16
x: 2, y: 16
x: 3, y: 15
x: 4, y: 14
x: 5, y: 13
x: 6, y: 13
x: 7, y: 12
x: 8, y: 11
x: 9, y: 10
x: 10, y: 9
x: 11, y: 8
x: 12, y: 7
x: 13, y: 7
x: 14, y: 6
Ruta obtenida
```

Figura 41: Archivo de rutas

Las coordenadas están dadas de acuerdo con el punto de partida, que está representado a partir del punto inicial en el simulador, hasta el destino, el cual es uno de los nutrientes en el simulador del Physarum y son de vital importancia para poder generar la ruta, ya que, con la ausencia de estos, no se puede generar una ruta. Las coordenadas están ordenadas de forma de aparición, lo que hace que haya una forma de seguirla correctamente y no se deba de perder, o seguir la ruta de manera errónea.

5.1.4. Ajustes del algoritmo basados en pruebas unitarias 1

Recopilando los datos obtenidos en algunas de las pruebas unitarias realizadas hasta el momento de realización de estas mismas, se aplicaron algunos cambios al programa, principalmente enfocados a mejorar cada una de sus funciones y que sea mucho más fácil en cuanto a la calidad del algoritmo.

Los cambios que han sido aplicados hasta el momento corresponden a los siguientes elementos que conforman el programa:

Implementación de la clase LoadMap: A pesar de que se pueda configurar un espacio en el lienzo por medio del dibujo, muchas veces al querer colocar un croquis o un mapa de determinado espacio, dibujar a mano con el ratón resulta poco conveniente, por lo que se implementó la clase LoadMap, la cual hace uso de la librería OpenCV, transformando una imagen en un dibujo en el lienzo del programa al inicializarse este último. Esto se puede ver en la Figura 42.

```

7  class LoadMap {
8      public:
9          void convertImageToMap(std::string);
10         void setDataToArray(int**, int, int);
11     private:
12         void grayscaleImage();
13         void setDataToRGBVector(int, int);
14     public:
15         float isImgProcessed = false;
16     private:
17         cv::Mat actualImage;
18         cv::Mat processedImage;
19         std::vector<std::vector<float>> rgbVector;
20         float threshold = 30;
21         int IMG_WIDTH = 500, IMG_HEIGHT = 500;
22 };

```

Figura 42: Implementación del LoadMap

```

8  class RandomRange {
9      public:
10         int getRandom(const int, const int);
11     private:
12     public:
13     private:
14
15 };

```

Figura 43: Clase RandomRange

Clase RandomRange: Al realizarse las primeras pruebas unitarias en el software, fue obtenido como resultado que es poco conveniente usar la función rand() debido a que tiende a ser mucho más lenta que otras implementaciones, además de que su aleatoriedad no es la mejor y además, no es multihilo, lo que impide que si en el futuro el algoritmo se quiere paralelizar, esto dificulte la implementación de número randoms que sean thread safe. Esto se puede ver en la Figura 43.

5.1.5. Ajustes del algoritmo basados en pruebas unitarias 2

En esta ocasión, no se realizaron cambios en el algoritmo, debido a que se pudo comprobar el correcto funcionamiento de este mismo y la cuestión real de funcionamiento o pruebas es el cambio de configuraciones en el estado inicial para la obtención de rutas, siendo evaluados distintos casos y configuraciones del estado inicial.

Debido a que hasta el momento el algoritmo cumple su función y las pruebas unitarias no revelaron algún resultado en el cual haya algún error o falla con alguna parte del algoritmo, en esta ocasión los ajustes al algoritmo no fueron realizados.

Es importante recalcar que el funcionamiento base del algoritmo es importante para la funcionalidad de la simulación en general, esto debido a que un pequeño cambio a alguna de sus partes puede alterar el resultado general y formar comportamientos o generar resultados no deseados y que no logran cumplir el objetivo general del software el cual es la generación de rutas a partir del organismo Physarum.

5.1.6. Diseño inicial de interfaz para control y monitorear

Para el diseño de la interfaz en la cual se puede controlar y a su vez, hacer uso del sistema de monitoreo del robot, es necesario conocer cada uno de los componentes que son necesarios mostrar y los que serán necesarios para realizar la funcionalidad correspondiente, los cuales corresponden a los siguientes:

Sistema de monitoreo: La interfaz debe mostrar la visión de la cámara que esta incorporada al robot, por lo que debe de haber un espacio dedicado a mostrar a esta misma.

Sistema LiDAR: En la interfaz también se debe mostrar cada uno de los puntos que son obtenidos a través de este sistema, los cuales representan los obstáculos que están siendo detectados por el robot, así como una representación de lo que se encuentra alrededor de éste.

Módulo de navegación: Debido a que el robot puede ser controlado a distancia, se necesita de la colocación de una serie de botones con los cuales el robot pueda realizar la navegación básica. Con los componentes mencionados anteriormente, se realiza el diseño de la interfaz inicial con la cual cada uno de estos es colocado de tal forma que se tenga acceso a él y se puedan realizar cada una de las tareas que fueron explicadas anteriormente.

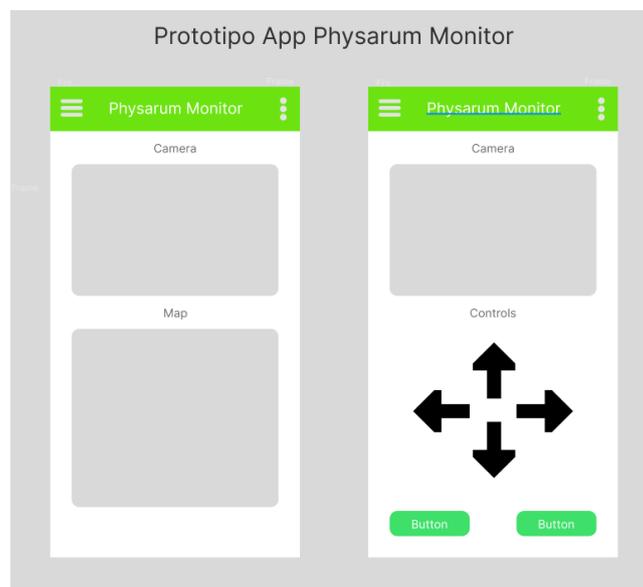


Figura 44: Prototipo App Physarum Monitor

Con la disposición mostrada en la Figura 44, podemos ver que se colocan los elementos que son necesarios para poder cumplir con cada una de las funciones que son necesarias para el control y monitoreo del robot. Se muestran dos configuraciones, una de las cuales contiene tanto la vista de la cámara, así como de la vista de la información que se nos es enviada por parte del robot, la cual es obtenida con el sensor LiDAR. La siguiente configuración contiene tanto la vista de la cámara del robot, como una serie de botones los cuales representan cada uno de los movimientos con los cuales es posible controlar al robot. Esta configuración en particular está diseñada para poder observar por medio de la cámara los movimientos que se están realizando a través de los botones y así, tener una mejor noción de lo que se está haciendo con el robot, así como para poder orientarse y poder tener un mejor control de este robot.

5.1.7. Ajustes de rutas basado en resultado de pruebas de aceptación 1

Con los resultados que fueron obtenidos a partir de la ejecución de las pruebas de aceptación, se toma en cuenta que hay algunas configuraciones o ajustes que se deben de hacer al algoritmo. Una de las primeras es generar una ruta 'mas limpia', es decir, una ruta en la cual existan mucho menos celdas las cuales impidan o dificulten generar una ruta un poco más optima o que causan

algunos impedimentos a la hora de generar el ordenamiento de coordenadas para ser enviadas al robot para la realización de su funcionalidad de llegar a su destino.

Una de las soluciones que se considera para poder obtener las rutas un poco más limpias y ajustarlas es limpiar un poco el camino o ruta generada por el Physarum una vez esta última siendo obtenida es eliminar las células que conforman al estado 8 y 5 respectivamente, las cuales no sean necesarias para el cálculo de la ruta o que no formen parte importante de la ruta que fue obtenida. Para esto se consideran dos casos importantes

- A) Las células alrededor de el punto inicial y el punto final.
- B) Las células que no conectan una célula con otra.

Una de las ideas para poder eliminar estas células las cuales dificultan la obtención correcta de las rutas es eliminar las células que están alrededor del estado inicial y el estado final, dejando únicamente las que tienen conexión con otras células las cuales pertenecen a la ruta general.

Esto se realizó de la siguiente forma creando un nuevo autómata con las siguientes reglas:

- Si el estado es 1 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 1.
- Si el estado es 2 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 2.
- Si el estado es 3, entonces es estado 7.
- Si el estado es 4 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 4.
- Si el estado es 5 y alrededor hay un punto inicial, entonces es estado 2, sino, si alrededor hay un nutriente, entonces es estado 4, sino es estado 1.
- Si es estado 6, entonces es estado 9.
- Si es estado 7, se queda en estado 7.
- Si el estado es 8 y alrededor hay un punto inicial, entonces es estado 2, sino, si alrededor hay un nutriente, entonces es estado 4, sino es estado 1.
- Si es estado 9, entonces se queda en estado 9.

Y queda plasmado con el siguiente Listing 3:

Listing 3: Ruta Ajuste 1

```
1         switch (tab[i][j]) {
2             case 1:
3                 if (!aroundState1)
4                     cellsAux[i][j] = 0;
5                 else
6                     cellsAux[i][j] = 1;
7             break;
```

```

8         case 2:
9             if (!aroundState1)
10                cellsAux[i][j] = 0;
11             else
12                cellsAux[i][j] = 2;
13         break;
14         case 3:
15             cellsAux[i][j] = 7;
16         break;
17         case 4:
18             if (!aroundState1)
19                cellsAux[i][j] = 0;
20             else
21                cellsAux[i][j] = 4;
22         break;
23         case 5:
24             cellsCounter++;
25             if (isInitialPointAround)
26                cellsAux[i][j] = 2;
27             else if (isNutrientAround)
28                cellsAux[i][j] = 4;
29             else
30                cellsAux[i][j] = 1;
31         break;
32         case 6:
33             cellsAux[i][j] = 9;
34         break;
35         case 7:
36             cellsAux[i][j] = 7;
37         break;
38         case 8:
39             cellsCounter++;
40             if (isInitialPointAround)
41                cellsAux[i][j] = 2;
42             else if(isNutrientAround)
43                cellsAux[i][j] = 4;
44             else
45                cellsAux[i][j] = 1;
46         break;
47         case 9:
48             cellsAux[i][j] = 9;
49         break;
50     }

```

Lo anterior se hace para poder obtener una ruta con una mejor claridad y que dificulte menos el ordenamiento y la manera en la cual se obtienen las coordenadas con las cuales se realiza el recorrido.

5.2. Robot Propuesto

Para construir el robot, se emplearán diversos materiales, cada uno con una función específica para asegurar la operatividad y eficiencia del dispositivo. A continuación, se detallan los materiales y sus descripciones.

El controlador para el motor paso a paso, Nema 23, será fundamental para manejar el movimiento del robot. Este componente incluye un controlador de motor a pasos que se utilizará en cuatro unidades para garantizar un control preciso de los motores. Los motores a pasos Nema

23 son conocidos por su precisión y confiabilidad, y en este caso, se utilizarán cuatro unidades, cada una con una placa frontal de 2.15 x 2.15 pulgadas (57 x 57 mm).

Para la estructura del robot, se utilizará una lámina de aluminio de calibre 14 (1.9 mm) con dimensiones de 20 cm x 40 cm, que proporcionará una base sólida y resistente. Además, se emplearán perfiles de aluminio 2040, específicamente de 20 x 40 mm y 500 mm de longitud, en dos unidades, para construir el marco del robot. También se utilizarán barras redondas sólidas de aluminio de 2 1/2" x 12", en cuatro unidades, para reforzar la estructura y proporcionar soporte adicional.

La movilidad del robot será posible gracias a las ruedas omnidireccionales de 6 pulgadas (152 mm) con rodamientos de silicona y cubos de aleación de aluminio, en cuatro unidades. Estas ruedas permitirán un movimiento fluido en múltiples direcciones. Además, se utilizarán diversos tornillos y tuercas, con un paquete de 60 unidades, para ensamblar todas las partes del robot de manera segura.

Para la energía, se utilizarán baterías de litio de 12V y 20000mAh, recargables, que proporcionarán la energía necesaria para la operación del robot. Se utilizarán dos de estas baterías. Un cargador de baterías de 14V y 20A, específico para baterías de litio de 12V, será empleado para mantener las baterías recargadas y operativas.

La electrónica del robot incluirá una Raspberry Pi 4 B, que actuará como el cerebro del dispositivo, gestionando las operaciones y los datos recibidos. Un sensor de distancia Detección y Rango de Luz (Light Detection and Ranging, LiDAR), modelo DTOF STL27L, permitirá al robot detectar obstáculos y medir distancias con precisión, utilizando un láser LiDAR 360° con bus Transmisor-Receptor Asíncrono Universal (Universal Asynchronous Receiver-Transmitter, UART) y un rango de 895 - 915 NM (tipo 905).

Para la visión, se empleará una cámara de visión nocturna de luz infrarroja de 5MP, con un ángulo de visión de 130-220 grados, específica para la Raspberry Pi 4B. También se incluirá un convertidor auto Boost Buck de CD-CD, de 5A y rango de 5V-30V, que ayudará a gestionar las diferentes necesidades de voltaje de los componentes electrónicos del robot.

Finalmente, se utilizará una lámina de acrílico transparente de 6 mm, con dimensiones de 60 x 120 cm, para crear cubiertas protectoras y otras partes visibles del robot. Este material es ideal por su durabilidad y resistencia a impactos.

Estos componentes, cuidadosamente seleccionados, se ensamblarán para crear un robot funcional, robusto y versátil, capaz de realizar diversas tareas con eficiencia y precisión.

Ahora claro enseñamos lo que sería la primera versión del sistema de control del robot propuesto.

5.2.1. Desarrollo inicial del sistema del control del robot

En esta subsección nos centramos en desarrollar el código inicial para poder controlar el robot, el cual se puede ver en el Listing 4.

Listing 4: Primera versión del código del sistema de control del robot

```
1 #include "CYdLidar.h"
```

```

2     #include <SFML/Graphics.hpp>
3     #include <opencv2/opencv.hpp>
4     #include <iostream>
5     #include <cstdlib>
6     #include <cstdio>
7     #include <pigpio.h>
8     #include <string>
9     #include <map>
10    #include <vector>
11    #include <atomic>
12    #include <thread>
13    #include <random>
14
15    using namespace std;
16    using namespace ydlidar;
17
18    // Pines y configuraci\on para los motores
19    const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
        motores
20    const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direcci\on
        para los motores
21    int frequency = 400; // Frecuencia inicial
22
23    std::atomic<bool> is_running(true);
24    std::atomic<bool> is_manual_mode(true);
25
26    void setMotorSpeed(int motor, int frequency) {
27        if (motor >= 0 && motor < 4) {
28            gpioSetPWmfrequency(PWM_PINS[motor], frequency);
29        }
30    }
31
32    void setMotorDirection(int motor, int direction) {
33        if (motor >= 0 && motor < 4) {
34            gpioWrite(DIR_PINS[motor], direction);
35        }
36    }
37
38    void stopMotors() {
39        for (int i = 0; i < 4; ++i) {
40            gpioPWM(PWM_PINS[i], 0);
41        }
42    }
43
44    void moveForward() {
45        for (int i = 0; i < 4; ++i) {
46            gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
47        }
48        setMotorDirection(0, 0); // Motor 1
49        setMotorDirection(2, 0); // Motor 3
50        setMotorDirection(1, 1); // Motor 2
51        setMotorDirection(3, 1); // Motor 4
52    }
53
54    void moveBackward() {
55        for (int i = 0; i < 4; ++i) {
56            gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
57        }

```

```

58         setMotorDirection(0, 1); // Motor 1
59         setMotorDirection(2, 1); // Motor 3
60         setMotorDirection(1, 0); // Motor 2
61         setMotorDirection(3, 0); // Motor 4
62     }
63
64     void turnLeft() {
65         for (int i = 0; i < 4; ++i) {
66             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
67                                     // al 50%
68         }
69         setMotorDirection(0, 1); // Motor 1
70         setMotorDirection(2, 1); // Motor 3
71         setMotorDirection(1, 1); // Motor 2
72         setMotorDirection(3, 1); // Motor 4
73     }
74
75     void turnRight() {
76         for (int i = 0; i < 4; ++i) {
77             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
78                                     // al 50%
79         }
80         setMotorDirection(0, 0); // Motor 1
81         setMotorDirection(2, 0); // Motor 3
82         setMotorDirection(1, 0); // Motor 2
83         setMotorDirection(3, 0); // Motor 4
84     }
85
86     void increaseSpeed() {
87         if (frequency < 1600) {
88             frequency += 100;
89             if (frequency > 1600) frequency = 1600;
90             for (int i = 0; i < 4; ++i) {
91                 setMotorSpeed(i, frequency);
92             }
93         }
94     }
95
96     void decreaseSpeed() {
97         if (frequency > 400) {
98             frequency -= 100;
99             if (frequency < 400) frequency = 400;
100            for (int i = 0; i < 4; ++i) {
101                setMotorSpeed(i, frequency);
102            }
103        }
104    }
105
106     sf::Color getPointColor(float distance, float maxRange) {
107         float ratio = distance / maxRange;
108         return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
109                                     // de rojo a verde
110     }
111
112     void randomMovement(CYdLidar &laser) {
113         std::random_device rd;
114         std::mt19937 gen(rd());
115         std::discrete_distribution<> dist({1, 10, 10, 70}); //
116                                     // Distribuci'on para la probabilidad de movimiento

```

```

114     while (is_running) {
115         if (!is_manual_mode) {
116             LaserScan scan;
117             if (laser.doProcessSimple(scan)) {
118                 bool obstacle_detected = false;
119                 for (const auto &point : scan.points) {
120                     if (point.range < 0.35) { // Detecta un obst
121                         \'aculo a 35 cm
122                         obstacle_detected = true;
123                         break;
124                     }
125                 }
126                 if (obstacle_detected) {
127                     // Si detecta un obst\'aculo, retrocede por 4
128                     segundos
129                     moveBackward();
130                     std::this_thread::sleep_for(std::chrono::
131                         seconds(4));
132                     stopMotors();
133
134                     // Luego gira aleatoriamente a la izquierda o
135                     derecha
136                     int turn = dist(gen) % 2;
137                     if (turn == 0) {
138                         turnLeft();
139                     } else {
140                         turnRight();
141                     }
142                     std::this_thread::sleep_for(std::chrono::
143                         seconds(2));
144                     stopMotors();
145                 } else {
146                     // Si no hay obst\'aculo, elige un movimiento
147                     aleatorio
148                     int move = dist(gen);
149                     switch (move) {
150                         case 0: moveBackward(); break;
151                         case 1: turnRight(); break;
152                         case 2: turnLeft(); break;
153                         case 3: moveForward(); break;
154                     }
155                 }
156                 std::this_thread::sleep_for(std::chrono::milliseconds
157                     (500));
158                 stopMotors();
159             }
160             std::this_thread::sleep_for(std::chrono::milliseconds(100)
161                 );
162         }
163     }
164
165     int main() {
166         // Inicializar pigpio
167         if (gpioInitialise() < 0) {
168             std::cerr << "Error: No se pudo inicializar pigpio." <<
169                 std::endl;
170             return -1;
171         }

```

```

165
166 // Configurar pines de direcci\on como salida
167 for (int i = 0; i < 4; ++i) {
168     gpioSetMode(DIR_PINS[i], PI_OUTPUT);
169     gpioSetMode(PWM_PINS[i], PI_OUTPUT);
170     setMotorSpeed(i, frequency); // Inicializar PWM con
        frecuencia inicial
171 }
172
173 // Aseg\urate de establecer XDG_RUNTIME_DIR
174 if (getenv("XDG_RUNTIME_DIR") == nullptr) {
175     setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
176 }
177
178 // Ejecuta libcamera-vid en un proceso separado y captura la
        salida en YUV, sin previsualizaci\on
179 FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
        nopreview -o -, "r");
180 if (!pipe) {
181     std::cerr << "Error: No se pudo ejecutar libcamera-vid."
        << std::endl;
182     return -1;
183 }
184
185 // Configura la ventana SFML
186 sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
        Visualization with LiDAR");
187 sf::Texture cameraTexture;
188 sf::Sprite cameraSprite;
189
190 // Buffer para leer los datos de video
191 const int width = 640;
192 const int height = 480;
193 std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
        el tama\~no del buffer para YUV420
194
195 cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
        data());
196 cv::Mat rgbImage(height, width, CV_8UC3);
197
198 std::string port;
199 ydlidar::os_init();
200
201 // Obtener los puertos disponibles de LiDAR
202 std::map<std::string, std::string> ports = ydlidar::
        lidarPortList();
203 if (ports.size() > 1) {
204     auto it = ports.begin();
205     std::advance(it, 1); // Selecciona el segundo puerto
        disponible
206     port = it->second;
207 } else if (ports.size() == 1) {
208     port = ports.begin()->second;
209 } else {
210     std::cerr << "No se detect\o ning\un LiDAR. Verifica la
        conexi\on." << std::endl;
211     return -1;
212 }
213
214 // Configuraci\on del LiDAR

```

```

215     int baudrate = 115200;
216     std::cout << "Baudrate: " << baudrate << std::endl;
217
218     CYdLidar laser;
219     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
        ());
220     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
        int));
221
222     bool isSingleChannel = true;
223     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
        sizeof(bool));
224
225     float max_range = 8.0f;
226     float min_range = 0.1f;
227     float max_angle = 180.0f;
228     float min_angle = -180.0f;
229     float frequency = 8.0f;
230
231     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
        );
232     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
        );
233     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
        );
234     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
        );
235     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
        float));
236
237     // Inicializar LiDAR
238     if (!laser.initialize()) {
239         std::cerr << "Error al inicializar el LiDAR." << std::endl
            ;
240         return -1;
241     }
242
243     // Iniciar el escaneo
244     if (!laser.turnOn()) {
245         std::cerr << "Error al encender el LiDAR." << std::endl;
246         return -1;
247     }
248
249     // Thread para movimiento aleatorio
250     std::thread randomMoveThread(randomMovement, std::ref(laser));
251
252     while (window.isOpen()) {
253         sf::Event event;
254         while (window.pollEvent(event)) {
255             if (event.type == sf::Event::Closed)
256                 window.close();
257
258             if (event.type == sf::Event::KeyPressed) {
259                 switch (event.key.code) {
260                     case sf::Keyboard::W:
261                         is_manual_mode = true;
262                         moveForward();
263                         break;
264                     case sf::Keyboard::S:
265                         is_manual_mode = true;

```

```

266         moveBackward();
267         break;
268     case sf::Keyboard::A:
269         is_manual_mode = true;
270         turnLeft();
271         break;
272     case sf::Keyboard::D:
273         is_manual_mode = true;
274         turnRight();
275         break;
276     case sf::Keyboard::Add:
277         is_manual_mode = true;
278         increaseSpeed();
279         break;
280     case sf::Keyboard::Subtract:
281         is_manual_mode = true;
282         decreaseSpeed();
283         break;
284     case sf::Keyboard::Space:
285         is_manual_mode = true;
286         stopMotors();
287         break;
288     case sf::Keyboard::K:
289         is_manual_mode = false;
290         break;
291     case sf::Keyboard::M:
292         is_manual_mode = true;
293         break;
294     default:
295         break;
296     }
297 }
298 }
299
300 // Leer los datos del video desde la tuber'ia
301 size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
302     pipe);
303 if (bytesRead != buffer.size()) {
304     std::cerr << "Error: No se pudo leer suficientes datos
305     de video." << std::endl;
306     continue;
307 }
308
309 // Convertir YUV420 a RGB
310 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
311
312 // Convertir a RGBA a~nadiendo un canal alfa
313 cv::Mat frame_rgba;
314 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
315
316 // Actualizar la textura de la c'amara con los datos del
317 frame
318 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows
319 )) {
320     std::cerr << "Error: No se pudo crear la textura." <<
321     std::endl;
322     continue;
323 }
324 cameraTexture.update(frame_rgba.ptr());

```

```

321     cameraSprite.setTexture(cameraTexture);
322     cameraSprite.setScale(
323         window.getSize().x / static_cast<float>(cameraTexture.
324             getSize().x),
325         window.getSize().y / static_cast<float>(cameraTexture.
326             getSize().y)
327     );
328
329     window.clear();
330     window.draw(cameraSprite);
331
332     // Crear un minimapa para el LiDAR
333     sf::RectangleShape minimap(sf::Vector2f(200, 200));
334     minimap.setFillColor(sf::Color(200, 200, 200, 150)); //
335     Fondo semitransparente
336     minimap.setPosition(10, 10); // Esquina superior izquierda
337
338     window.draw(minimap);
339
340     // Dibujar el centro del LiDAR (color azul)
341     sf::CircleShape lidarCenter(5); // Radio del círculo del
342     LiDAR
343     lidarCenter.setFillColor(sf::Color::Blue);
344     lidarCenter.setPosition(105, 105); // Posición del
345     centro en el minimapa
346
347     window.draw(lidarCenter);
348
349     // Dibujar la línea hacia el norte
350     sf::Vertex line[] =
351     {
352         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
353         sf::Vertex(sf::Vector2f(110, 60), sf::Color::Black) //
354         Línea hacia arriba (norte)
355     };
356
357     window.draw(line, 2, sf::Lines);
358
359     LaserScan scan;
360     if (laser.doProcessSimple(scan)) {
361         for (const auto& point : scan.points) {
362             // Convertir coordenadas polares a cartesianas
363             float x = point.range * cos(point.angle);
364             float y = point.range * sin(point.angle);
365
366             // Ajustar los puntos al minimapa
367             float scale = 25.0f;
368             float adjustedX = 110 + x * scale;
369             float adjustedY = 110 - y * scale; // Invertir Y
370             para coordinar con la pantalla
371
372             // Dibujar los puntos en el minimapa, excluyendo
373             el centro (0,0)
374             if (point.range > 0.05) {
375                 sf::CircleShape lidarPoint(2);
376                 lidarPoint.setPosition(adjustedX, adjustedY);
377                 lidarPoint.setFillColor(getPointColor(point.
378                     range, max_range));
379
380                 window.draw(lidarPoint);

```

```

372         }
373     }
374     } else {
375         std::cerr << "No se pudieron obtener los datos del
376         LiDAR." << std::endl;
377     }
378     window.display();
379 }
380
381 // Detener el escaneo del LiDAR
382 laser.turnOff();
383 laser.disconnecting();
384
385 // Cierra la tuber\`ia, detiene los motores y apaga el robot
386 pclose(pipe);
387 stopMotors();
388 gpioTerminate();
389
390 is_running = false;
391 randomMoveThread.join();
392
393 return 0;
394 }

```

En el Listing 4 se inicializan los motores y se establecen las funciones para controlar el movimiento del robot. Además, se configura el LiDAR y se inicia el escaneo. El robot se mueve aleatoriamente y evita obstáculos detectados por el LiDAR. La cámara muestra la vista del robot y el minimapa muestra los puntos detectados por el LiDAR. El usuario puede controlar manualmente el robot con las teclas W, A, S y D para moverse hacia adelante, izquierda, atrás y derecha, respectivamente. Las teclas V y B aumentan y disminuyen la velocidad del robot, respectivamente. La tecla Espacio detiene el robot. La tecla K activa el modo automático y la tecla M activa el modo manual. El robot se mueve aleatoriamente y evita obstáculos detectados por el LiDAR. El código se ejecuta en un bucle hasta que se cierra la ventana de la cámara. Al final, se detiene el escaneo del LiDAR y se apaga el robot.

5.2.2. Ajustes de código en función de pruebas unitarias y de aceptación 1

El código del robot funcionó correctamente en las pruebas unitarias y de aceptación, con la excepción del modo aleatorio, que presentó problemas de movimiento. El robot se movía de manera ineficiente en el modo aleatorio, debido a la forma en que se le ordenaba moverse y que en 0,0 detectaba obstáculos donde no los había, en otras palabras daba falsos positivos. Por lo tanto, se realizaron ajustes en el código para mejorar su desempeño y eficiencia, los cuales se pueden ver en el Listing 5.

Listing 5: Primer ajuste de código

```

1     #include "CYdLidar.h"
2     #include <SFML/Graphics.hpp>
3     #include <opencv2/opencv.hpp>
4     #include <iostream>
5     #include <cstdlib>
6     #include <cstdio>
7     #include <pigpio.h>
8     #include <string>
9     #include <map>
10    #include <vector>

```

```

11  #include <atomic>
12  #include <thread>
13  #include <random>
14  #include <sys/socket.h>
15  #include <sys/un.h>
16  #include <unistd.h>
17  #include <cstring>
18  #include <pthread.h>
19  #include <vector>
20  #include <mutex>
21
22  using namespace std;
23  using namespace ydlidar;
24
25  // Pines y configuracin para los motores
26  const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
      motores
27  const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin para
      los motores
28  int frequency = 400; // Frecuencia inicial
29
30  std::atomic<bool> is_running(true);
31  std::atomic<bool> is_manual_mode(true);
32
33  void setMotorSpeed(int motor, int frequency) {
34      if (motor >= 0 && motor < 4) {
35          gpioSetPWmfrequency(PWM_PINS[motor], frequency);
36      }
37  }
38
39  void setMotorDirection(int motor, int direction) {
40      if (motor >= 0 && motor < 4) {
41          gpioWrite(DIR_PINS[motor], direction);
42      }
43  }
44
45  void stopMotors() {
46      for (int i = 0; i < 4; ++i) {
47          gpioPWM(PWM_PINS[i], 0);
48      }
49  }
50
51  void moveForward() {
52      for (int i = 0; i < 4; ++i) {
53          gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
      50%
54      }
55      setMotorDirection(0, 0); // Motor 1
56      setMotorDirection(2, 0); // Motor 3
57      setMotorDirection(1, 1); // Motor 2
58      setMotorDirection(3, 1); // Motor 4
59  }
60
61  void moveBackward() {
62      for (int i = 0; i < 4; ++i) {
63          gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
      50%
64      }
65      setMotorDirection(0, 1); // Motor 1
66      setMotorDirection(2, 1); // Motor 3

```

```

67     setMotorDirection(1, 0); // Motor 2
68     setMotorDirection(3, 0); // Motor 4
69 }
70
71 void turnLeft() {
72     for (int i = 0; i < 4; ++i) {
73         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
74             50%
75     }
76     setMotorDirection(0, 1); // Motor 1
77     setMotorDirection(2, 1); // Motor 3
78     setMotorDirection(1, 1); // Motor 2
79     setMotorDirection(3, 1); // Motor 4
80 }
81 void turnRight() {
82     for (int i = 0; i < 4; ++i) {
83         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
84             50%
85     }
86     setMotorDirection(0, 0); // Motor 1
87     setMotorDirection(2, 0); // Motor 3
88     setMotorDirection(1, 0); // Motor 2
89     setMotorDirection(3, 0); // Motor 4
90 }
91 void increaseSpeed() {
92     if (frequency < 2000) {
93         frequency += 100;
94         if (frequency > 2000) frequency = 2000;
95         for (int i = 0; i < 4; ++i) {
96             setMotorSpeed(i, frequency);
97         }
98     }
99 }
100
101 void decreaseSpeed() {
102     if (frequency > 400) {
103         frequency -= 100;
104         if (frequency < 400) frequency = 400;
105         for (int i = 0; i < 4; ++i) {
106             setMotorSpeed(i, frequency);
107         }
108     }
109 }
110
111 sf::Color getPointColor(float distance, float maxRange) {
112     float ratio = distance / maxRange;
113     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de
114         rojo a verde
115 }
116
117 int contadorObstaculoTotal = 0;
118 int obstaculoHola = 0;
119 int obsRelativo = 0;
120 void randomMovement(CYdLidar &laser) {
121     std::random_device rd;
122     std::mt19937 gen(rd());

```

```

123     std::discrete_distribution<> dist({1, 0, 10, 70}); // Distribucion
        para la probabilidad de movimiento
124
125     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
        grados en radianes
126     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
        grados en radianes
127     const float DETECTION_RADIUS = 0.25f; // 35 cm
128
129     std::vector<float> previousScanPoints;
130
131     while (is_running) {
132         if (!is_manual_mode) {
133             LaserScan scan;
134             if (laser.doProcessSimple(scan)) {
135                 bool obstacle_detected = false;
136                 std::vector<float> currentScanPoints;
137
138                 for (const auto &point : scan.points) {
139                     // Convertir el ngulo del punto al ngulo relativo
                        al "sur" del robot
140                     float adjusted_angle = point.angle + M_PI;
141                     //std::cout << "Er " << point.range << std::endl;
142                     // Verificar si el punto est dentro del rango
                        frontal de 30
143                     if (point.range > 0 && point.range < 0.40 && abs(
                            point.angle) < adjusted_angle) {
144                         obstacle_detected = true;
145                         currentScanPoints.push_back(point.range);
146                         std::cout << "Obstacle distance: " << (float)
                            point.range << " Y en el angulo " << point
                                .angle << std::endl;
147                         break;
148                     }
149                 }
150
151                 if (obstacle_detected) {
152                     // Si detecta un obstculo, retrocede por 4
                        segundos
153                     contadorObstaculoTotal++;
154                     std::cout << "Obs: " << contadorObstaculoTotal <<
                            std::endl;
155                     //obsRelativo++;
156                     stopMotors();
157                     //Luego gira aleatoriamente a la izquierda o
                        derecha
158                     int turn = dist(gen) % 2;
159                     if (turn == 0) {
160                         turnLeft();
161                         std::this_thread::sleep_for(std::chrono::
                            seconds(5));
162                     } else {
163                         turnRight();
164                         std::this_thread::sleep_for(std::chrono::
                            seconds(5));
165                     }
166                     std::this_thread::sleep_for(std::chrono::seconds
                            (2));
167                     stopMotors();
168

```

```

169         if(!previousScanPoints.empty() &&
           previousScanPoints.size() == currentScanPoints
           .size()){
170             bool mismoObs = true;
171             for(size_t i = 0; i < currentScanPoints.
           size(); ++i){
172                 if(fabs(currentScanPoints[i] -
           previousScanPoints[i]) > 0.05){
173                     mismoObs = false;
174                     break;
175                 }
176             }
177             if(mismoObs){
178                 obsRelativo++;
179             }else{
180                 obsRelativo = 0;
181             }
182         }
183
184         previousScanPoints = currentScanPoints;
185
186         if((obsRelativo > 3)){
187             //obsRelativo = 0;
188             moveBackward();
189             std::this_thread::sleep_for(std::chrono::
           seconds(3));
190             stopMotors();
191             int turn = dist(gen) % 2;
192             if (turn == 0) {
193                 turnLeft();
194                 std::this_thread::sleep_for(std::chrono::
           seconds(5));
195             } else {
196                 turnRight();
197                 std::this_thread::sleep_for(std::chrono::
           seconds(5));
198             }
199             std::this_thread::sleep_for(std::chrono::
           seconds(2));
200             stopMotors();
201         }
202     } else {
203         moveForward();
204         // Si no hay obstculo, elige un movimiento
           aleatorio
205         //
206         //int move = dist(gen);
207         //switch (move) {
208             // case 0: moveBackward(); break;
209             // case 1: turnRight(); break;
210             // case 2: turnLeft(); break;
211             // case 3: moveForward(); break;
212         //}
213     }
214 }
215
216     std::this_thread::sleep_for(std::chrono::milliseconds(500)
           );
217     //stopMotors();
218 }

```

```

219         std::this_thread::sleep_for(std::chrono::milliseconds(100));
220     }
221 }
222
223 int main() {
224     // Inicializar pigpio
225     if (gpioInitialise() < 0) {
226         std::cerr << "Error: No se pudo inicializar pigpio." << std::
                endl;
227         return -1;
228     }
229
230     // Configurar pines de direccin como salida
231     for (int i = 0; i < 4; ++i) {
232         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
233         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
234         setMotorSpeed(i, frequency); // Inicializar PWM con
                frecuencia inicial
235     }
236
237     // Asegrate de establecer XDG_RUNTIME_DIR
238     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
239         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
240     }
241
242     // Ejecuta libcamera-vid en un proceso separado y captura la
                salida en YUV, sin previsualizacn
243     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview
                -o -", "r");
244     if (!pipe) {
245         std::cerr << "Error: No se pudo ejecutar libcamera-vid." <<
                std::endl;
246         return -1;
247     }
248
249     // Configura la ventana SFML
250     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
                Visualization with LiDAR");
251     sf::Texture cameraTexture;
252     sf::Sprite cameraSprite;
253
254     // Buffer para leer los datos de video
255     const int width = 640;
256     const int height = 480;
257     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
                tamao del buffer para YUV420
258
259     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data
                ());
260     cv::Mat rgbImage(height, width, CV_8UC3);
261
262     std::string port;
263     ydllidar::os_init();
264
265     // Obtener los puertos disponibles de LiDAR
266     std::map<std::string, std::string> ports = ydllidar::lidarPortList
                ();
267     if (ports.size() > 1) {
268         auto it = ports.begin();

```

```

269         std::advance(it, 1); // Selecciona el segundo puerto
           disponible
270         port = it->second;
271     } else if (ports.size() == 1) {
272         port = ports.begin()->second;
273     } else {
274         std::cerr << "No se detect ningun LiDAR. Verifica la conexin."
           << std::endl;
275         return -1;
276     }
277
278     // Configuracin del LiDAR
279     int baudrate = 115200;
280     std::cout << "Baudrate: " << baudrate << std::endl;
281
282     CYdLidar laser;
283     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
284     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int))
           ;
285
286     bool isSingleChannel = true;
287     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof
           (bool));
288
289     float max_range = 8.0f;
290     float min_range = 0.1f;
291     float max_angle = 180.0f;
292     float min_angle = -180.0f;
293     float frequency = 8.0f;
294
295     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
296     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
297     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
298     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
299     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float
           ));
300
301     // Inicializar LiDAR
302     if (!laser.initialize()) {
303         std::cerr << "Error al inicializar el LiDAR." << std::endl;
304         return -1;
305     }
306
307     // Iniciar el escaneo
308     if (!laser.turnOn()) {
309         std::cerr << "Error al encender el LiDAR." << std::endl;
310         return -1;
311     }
312
313     // Thread para movimiento aleatorio
314     std::thread randomMoveThread(randomMovement, std::ref(laser));
315
316     while (window.isOpen()) {
317         sf::Event event;
318         while (window.pollEvent(event)) {
319             if (event.type == sf::Event::Closed)
320                 window.close();
321
322             if (event.type == sf::Event::KeyPressed) {
323                 switch (event.key.code) {

```

```

324         case sf::Keyboard::W:
325             is_manual_mode = true;
326             moveForward();
327             break;
328         case sf::Keyboard::S:
329             is_manual_mode = true;
330             moveBackward();
331             break;
332         case sf::Keyboard::A:
333             is_manual_mode = true;
334             turnLeft();
335             break;
336         case sf::Keyboard::D:
337             is_manual_mode = true;
338             turnRight();
339             break;
340         case sf::Keyboard::V:
341             is_manual_mode = true;
342             increaseSpeed();
343             break;
344         case sf::Keyboard::B:
345             is_manual_mode = true;
346             decreaseSpeed();
347             break;
348         case sf::Keyboard::Space:
349             is_manual_mode = true;
350             stopMotors();
351             break;
352         case sf::Keyboard::K:
353             is_manual_mode = false;
354             break;
355         case sf::Keyboard::M:
356             is_manual_mode = true;
357             break;
358         default:
359             break;
360     }
361 }
362 }
363
364 // Leer los datos del video desde la tubera
365 size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe
366 );
367 if (bytesRead != buffer.size()) {
368     std::cerr << "Error: No se pudo leer suficientes datos de
369     video." << std::endl;
370     continue;
371 }
372
373 // Convertir YUV420 a RGB
374 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
375
376 // Convertir a RGBA aadiendo un canal alfa
377 cv::Mat frame_rgba;
378 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
379
380 // Actualizar la textura de la cmara con los datos del frame
381 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
382     std::cerr << "Error: No se pudo crear la textura." << std
383     ::endl;

```

```

381         continue;
382     }
383     cameraTexture.update(frame_rgba.ptr());
384
385     cameraSprite.setTexture(cameraTexture);
386     cameraSprite.setScale(
387         window.getSize().x / static_cast<float>(cameraTexture.
388             getSize().x),
389         window.getSize().y / static_cast<float>(cameraTexture.
390             getSize().y)
391     );
392
393     window.clear();
394     window.draw(cameraSprite);
395
396     // Crear un minimapa para el LiDAR
397     sf::RectangleShape minimap(sf::Vector2f(200, 200));
398     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
399     semitransparente
400     minimap.setPosition(10, 10); // Esquina superior izquierda
401
402     window.draw(minimap);
403
404     // Dibujar el centro del LiDAR (color azul)
405     sf::CircleShape lidarCenter(5); // Radio del crculo del LiDAR
406     lidarCenter.setFillColor(sf::Color::Blue);
407     lidarCenter.setPosition(105, 105); // Posicin del centro en el
408     minimapa
409
410     window.draw(lidarCenter);
411
412     // Dibujar la lnea hacia el norte
413     sf::Vertex line[] =
414     {
415         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
416         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) //
417         Lnea hacia arriba (norte)
418     };
419
420     window.draw(line, 2, sf::Lines);
421
422     LaserScan scan;
423     if (laser.doProcessSimple(scan)) {
424         for (const auto& point : scan.points) {
425             // Convertir coordenadas polares a cartesianas
426             float x = point.range * cos(point.angle);
427             float y = point.range * sin(point.angle);
428
429             // Ajustar los puntos al minimapa
430             float scale = 25.0f;
431             float adjustedX = 110 + x * scale;
432             float adjustedY = 110 - y * scale; // Invertir Y para
433             coordinar con la pantalla
434
435             // Dibujar los puntos en el minimapa, excluyendo el
436             centro (0,0)
437             if (point.range > 0.05) {
438                 sf::CircleShape lidarPoint(2);
439                 lidarPoint.setPosition(adjustedX, adjustedY);

```

```

433         lidarPoint.setFillColor(getPointColor(point.range,
434             max_range));
435         window.draw(lidarPoint);
436     }
437     //std::cout << "An " << point.range <<std :: endl;
438 }
439 } else {
440     std::cerr << "No se pudieron obtener los datos del LiDAR."
441     << std::endl;
442 }
443     window.display();
444 }
445
446 // Detener el escaneo del LiDAR
447 laser.turnOff();
448 laser.disconnecting();
449
450 // Cierra la tubera, detiene los motores y apaga el robot
451 pclose(pipe);
452 stopMotors();
453 gpioTerminate();
454
455 is_running = false;
456 randomMoveThread.join();
457
458 return 0;
459 }

```

5.2.3. Ajustes de código en función de pruebas unitarias y de aceptación 2

En este caso se agregaron modificaciones para el mejor funcionamiento del detectar obstáculos sobre todo los ángulos de giro y la distancia de detección de los obstáculos. Estas modificaciones se realizaron en el código principal del robot, el cual se muestra en el Listing 6.

Listing 6: Segundo ajuste de código

```

1     #include "CYdLidar.h"
2     #include <SFML/Graphics.hpp>
3     #include <opencv2/opencv.hpp>
4     #include <iostream>
5     #include <cstdlib>
6     #include <cstdio>
7     #include <pigpio.h>
8     #include <string>
9     #include <map>
10    #include <vector>
11    #include <atomic>
12    #include <thread>
13    #include <random>
14    #include <sys/socket.h>
15    #include <sys/un.h>
16    #include <unistd.h>
17    #include <cstring>
18    #include <pthread.h>
19    #include <vector>
20    #include <mutex>
21

```

```

22     using namespace std;
23     using namespace ydlidar;
24
25     // Pines y configuracin para los motores
26     const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
        motores
27     const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin
        para los motores
28     int frequency = 400; // Frecuencia inicial
29
30     std::atomic<bool> is_running(true);
31     std::atomic<bool> is_manual_mode(true);
32
33     void setMotorSpeed(int motor, int frequency) {
34         if (motor >= 0 && motor < 4) {
35             gpioSetPWmfrequency(PWM_PINS[motor], frequency);
36         }
37     }
38
39     void setMotorDirection(int motor, int direction) {
40         if (motor >= 0 && motor < 4) {
41             gpioWrite(DIR_PINS[motor], direction);
42         }
43     }
44
45     void stopMotors() {
46         for (int i = 0; i < 4; ++i) {
47             gpioPWM(PWM_PINS[i], 0);
48         }
49     }
50
51     void moveForward() {
52         for (int i = 0; i < 4; ++i) {
53             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
54         }
55         setMotorDirection(0, 0); // Motor 1
56         setMotorDirection(2, 0); // Motor 3
57         setMotorDirection(1, 1); // Motor 2
58         setMotorDirection(3, 1); // Motor 4
59     }
60
61     void moveBackward() {
62         for (int i = 0; i < 4; ++i) {
63             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
64         }
65         setMotorDirection(0, 1); // Motor 1
66         setMotorDirection(2, 1); // Motor 3
67         setMotorDirection(1, 0); // Motor 2
68         setMotorDirection(3, 0); // Motor 4
69     }
70
71     void turnLeft() {
72         for (int i = 0; i < 4; ++i) {
73             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
74         }
75         setMotorDirection(0, 1); // Motor 1
76         setMotorDirection(2, 1); // Motor 3

```

```

77         setMotorDirection(1, 1); // Motor 2
78         setMotorDirection(3, 1); // Motor 4
79     }
80
81     void turnRight() {
82         for (int i = 0; i < 4; ++i) {
83             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
84                                     // al 50%
85         }
86         setMotorDirection(0, 0); // Motor 1
87         setMotorDirection(2, 0); // Motor 3
88         setMotorDirection(1, 0); // Motor 2
89         setMotorDirection(3, 0); // Motor 4
90     }
91
92     void increaseSpeed() {
93         if (frequency < 2000) {
94             frequency += 100;
95             if (frequency > 2000) frequency = 2000;
96             for (int i = 0; i < 4; ++i) {
97                 setMotorSpeed(i, frequency);
98             }
99         }
100     }
101
102     void decreaseSpeed() {
103         if (frequency > 400) {
104             frequency -= 100;
105             if (frequency < 400) frequency = 400;
106             for (int i = 0; i < 4; ++i) {
107                 setMotorSpeed(i, frequency);
108             }
109         }
110     }
111
112     sf::Color getPointColor(float distance, float maxRange) {
113         float ratio = distance / maxRange;
114         return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
115                                     // de rojo a verde
116     }
117
118     int contadorObstaculoTotal = 0;
119     int obstaculoHola = 0;
120     int obsRelativo = 0;
121     void randomMovement(CYdLidar &laser) {
122         std::random_device rd;
123         std::mt19937 gen(rd());
124         std::discrete_distribution<> dist({1, 0, 10, 70}); //
125                                     // Distribucion para la probabilidad de movimiento
126
127         const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
128                                     // grados en radaianes
129         const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
130                                     // grados en radianes
131         const float DETECTION_RADIUS = 0.25f; // 35 cm
132
133         std::vector<float> previousScanPoints;
134
135         while (is_running) {

```

```

132     if (!is_manual_mode) {
133         LaserScan scan;
134         if (laser.doProcessSimple(scan)) {
135             bool obstacle_detected = false;
136             std::vector<float> currentScanPoints;
137
138             for (const auto &point : scan.points) {
139                 // Convertir el ngulo del punto al ngulo
140                 // relativo al "sur" del robot
141                 float adjusted_angle = point.angle + M_PI;
142                 //std::cout << "Er " << point.range << std::
143                 // endl;
144                 // Verificar si el punto est dentro del rango
145                 // frontal de 30
146                 if (point.range > 0 &&point.range < 0.40 &&
147                     abs(point.angle) < adjusted_angle) {
148                     obstacle_detected = true;
149                     currentScanPoints.push_back(point.range);
150                     std::cout << "Obstacle distance: " << (
151                         float)point.range << " Y en el angulo
152                         "<<point.angle << std::endl;
153                     break;
154                 }
155             }
156
157             if (obstacle_detected) {
158                 // Si detecta un obstaculo, retrocede por 4
159                 // segundos
160                 contadorObstaculoTotal++;
161                 std::cout << "Obs: " << contadorObstaculoTotal
162                     << std::endl;
163                 //obsRelativo++;
164                 stopMotors();
165                 //Luego gira aleatoriamente a la izquierda o
166                 // derecha
167                 int turn = dist(gen) % 2;
168                 if (turn == 0) {
169                     turnLeft();
170                     std::this_thread::sleep_for(std::chrono::
171                         seconds(5));
172                 } else {
173                     turnRight();
174                     std::this_thread::sleep_for(std::chrono::
175                         seconds(5));
176                 }
177                 std::this_thread::sleep_for(std::chrono::
178                     seconds(2));
179                 stopMotors();
180
181                 if(!previousScanPoints.empty() &&
182                     previousScanPoints.size() ==
183                     currentScanPoints.size()){
184                     bool mismoObs = true;
185                     for(size_t i = 0; i <
186                         currentScanPoints.size(); ++i){
187                         if(fabs(currentScanPoints[i] -
188                             previousScanPoints[i]) > 0.05)
189                             {
190                             mismoObs = false;
191                             break;

```

```

175         }
176     }
177     if(mismoObs){
178         obsRelativo++;
179     }else{
180         obsRelativo = 0;
181     }
182 }
183
184 previousScanPoints = currentScanPoints;
185
186 if((obsRelativo > 3)){
187     //obsRelativo = 0;
188     moveBackward();
189     std::this_thread::sleep_for(std::chrono::
190         seconds(3));
191     stopMotors();
192     int turn = dist(gen) % 2;
193     if (turn == 0) {
194         turnLeft();
195         std::this_thread::sleep_for(std::
196             chrono::seconds(5));
197     } else {
198         turnRight();
199         std::this_thread::sleep_for(std::
200             chrono::seconds(5));
201     }
202     std::this_thread::sleep_for(std::
203         chrono::seconds(2));
204     stopMotors();
205 }
206 } else {
207     moveForward();
208     // Si no hay obstculo, elige un movimiento
209     // aleatorio
210     //
211     //int move = dist(gen);
212     //switch (move) {
213     //    // case 0: moveBackward(); break;
214     //    // case 1: turnRight(); break;
215     //    // case 2: turnLeft(); break;
216     //    // case 3: moveForward(); break;
217     //}
218 }
219 }
220
221     std::this_thread::sleep_for(std::chrono::milliseconds
222         (500));
223     //stopMotors();
224 }
225     std::this_thread::sleep_for(std::chrono::milliseconds(100)
226         );
227 }
228 }
229
230 int main() {
231     // Inicializar pigpio
232     if (gpioInitialise() < 0) {
233         std::cerr << "Error: No se pudo inicializar pigpio." <<
234             std::endl;

```

```

227         return -1;
228     }
229
230     // Configurar pines de direccin como salida
231     for (int i = 0; i < 4; ++i) {
232         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
233         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
234         setMotorSpeed(i, frequency); // Inicializar PWM con
           frecuencia inicial
235     }
236
237     // Asegrate de establecer XDG_RUNTIME_DIR
238     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
239         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
240     }
241
242     // Ejecuta libcamera-vid en un proceso separado y captura la
           salida en YUV, sin previsualizacn
243     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
           nopreview -o -, "r");
244     if (!pipe) {
245         std::cerr << "Error: No se pudo ejecutar libcamera-vid."
           << std::endl;
246         return -1;
247     }
248
249     // Configura la ventana SFML
250     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
           Visualization with LiDAR");
251     sf::Texture cameraTexture;
252     sf::Sprite cameraSprite;
253
254     // Buffer para leer los datos de video
255     const int width = 640;
256     const int height = 480;
257     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
           el tamao del buffer para YUV420
258
259     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
           data());
260     cv::Mat rgbImage(height, width, CV_8UC3);
261
262     std::string port;
263     ydlidar::os_init();
264
265     // Obtener los puertos disponibles de LiDAR
266     std::map<std::string, std::string> ports = ydlidar::
           lidarPortList();
267     if (ports.size() > 1) {
268         auto it = ports.begin();
269         std::advance(it, 1); // Selecciona el segundo puerto
           disponible
270         port = it->second;
271     } else if (ports.size() == 1) {
272         port = ports.begin()->second;
273     } else {
274         std::cerr << "No se detect ningn LiDAR. Verifica la
           conexin." << std::endl;
275         return -1;
276     }

```

```

277
278 // Configuracin del LiDAR
279 int baudrate = 115200;
280 std::cout << "Baudrate: " << baudrate << std::endl;
281
282 CYdLidar laser;
283 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
    ());
284 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
    int));
285
286 bool isSingleChannel = true;
287 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
    sizeof(bool));
288
289 float max_range = 8.0f;
290 float min_range = 0.1f;
291 float max_angle = 180.0f;
292 float min_angle = -180.0f;
293 float frequency = 8.0f;
294
295 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
    );
296 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
    );
297 laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
    );
298 laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
    );
299 laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
    float));
300
301 // Inicializar LiDAR
302 if (!laser.initialize()) {
303     std::cerr << "Error al inicializar el LiDAR." << std::endl
        ;
304     return -1;
305 }
306
307 // Iniciar el escaneo
308 if (!laser.turnOn()) {
309     std::cerr << "Error al encender el LiDAR." << std::endl;
310     return -1;
311 }
312
313 // Thread para movimiento aleatorio
314 std::thread randomMoveThread(randomMovement, std::ref(laser));
315
316 while (window.isOpen()) {
317     sf::Event event;
318     while (window.pollEvent(event)) {
319         if (event.type == sf::Event::Closed)
320             window.close();
321
322         if (event.type == sf::Event::KeyPressed) {
323             switch (event.key.code) {
324                 case sf::Keyboard::W:
325                     is_manual_mode = true;
326                     moveForward();
327                     break;

```

```

328         case sf::Keyboard::S:
329             is_manual_mode = true;
330             moveBackward();
331             break;
332         case sf::Keyboard::A:
333             is_manual_mode = true;
334             turnLeft();
335             break;
336         case sf::Keyboard::D:
337             is_manual_mode = true;
338             turnRight();
339             break;
340         case sf::Keyboard::V:
341             is_manual_mode = true;
342             increaseSpeed();
343             break;
344         case sf::Keyboard::B:
345             is_manual_mode = true;
346             decreaseSpeed();
347             break;
348         case sf::Keyboard::Space:
349             is_manual_mode = true;
350             stopMotors();
351             break;
352         case sf::Keyboard::K:
353             is_manual_mode = false;
354             break;
355         case sf::Keyboard::M:
356             is_manual_mode = true;
357             break;
358         default:
359             break;
360     }
361 }
362 }
363
364 // Leer los datos del video desde la tubera
365 size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
366     pipe);
367 if (bytesRead != buffer.size()) {
368     std::cerr << "Error: No se pudo leer suficientes datos
369     de video." << std::endl;
370     continue;
371 }
372
373 // Convertir YUV420 a RGB
374 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
375
376 // Convertir a RGBA aadiendo un canal alfa
377 cv::Mat frame_rgba;
378 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
379
380 // Actualizar la textura de la cmara con los datos del
381 frame
382 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows
383 )) {
384     std::cerr << "Error: No se pudo crear la textura." <<
385     std::endl;
386     continue;
387 }

```

```

383     cameraTexture.update(frame_rgba.ptr());
384
385     cameraSprite.setTexture(cameraTexture);
386     cameraSprite.setScale(
387         window.getSize().x / static_cast<float>(cameraTexture.
388             getSize().x),
389         window.getSize().y / static_cast<float>(cameraTexture.
390             getSize().y)
391     );
392
393     window.clear();
394     window.draw(cameraSprite);
395
396     // Crear un minimapa para el LiDAR
397     sf::RectangleShape minimap(sf::Vector2f(200, 200));
398     minimap.setFill-color(sf::Color(200, 200, 200, 150)); //
399     Fondo semitransparente
400     minimap.setPosition(10, 10); // Esquina superior izquierda
401
402     window.draw(minimap);
403
404     // Dibujar el centro del LiDAR (color azul)
405     sf::CircleShape lidarCenter(5); // Radio del circulo del
406     LiDAR
407     lidarCenter.setFill-color(sf::Color::Blue);
408     lidarCenter.setPosition(105, 105); // Posicin del centro
409     en el minimapa
410
411     window.draw(lidarCenter);
412
413     // Dibujar la linea hacia el norte
414     sf::Vertex line[] =
415     {
416         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
417         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black)
418         // Linea hacia arriba (norte)
419     };
420
421     window.draw(line, 2, sf::Lines);
422
423     LaserScan scan;
424     if (laser.doProcessSimple(scan)) {
425         for (const auto& point : scan.points) {
426             // Convertir coordenadas polares a cartesianas
427             float x = point.range * cos(point.angle);
428             float y = point.range * sin(point.angle);
429
430             // Ajustar los puntos al minimapa
431             float scale = 25.0f;
432             float adjustedX = 110 + x * scale;
433             float adjustedY = 110 - y * scale; // Invertir Y
434             para coordinar con la pantalla
435
436             // Dibujar los puntos en el minimapa, excluyendo
437             el centro (0,0)
438             if (point.range > 0.05) {
439                 sf::CircleShape lidarPoint(2);
440                 lidarPoint.setPosition(adjustedX, adjustedY);
441                 lidarPoint.setFill-color(getPointColor(point.
442                     range, max_range));

```

```

434
435         window.draw(lidarPoint);
436     }
437     //std::cout << "An " << point.range <<std :: endl;
438 }
439 } else {
440     std::cerr << "No se pudieron obtener los datos del
LiDAR." << std::endl;
441 }
442
443     window.display();
444 }
445
446     // Detener el escaneo del LiDAR
447     laser.turnOff();
448     laser.disconnecting();
449
450     // Cierra la tubera, detiene los motores y apaga el robot
451     pclose(pipe);
452     stopMotors();
453     gpioTerminate();
454
455     is_running = false;
456     randomMoveThread.join();
457
458     return 0;
459 }

```

5.2.4. Ajustes de código en función de las pruebas de carga 1

Las pruebas de carga realizadas al robot propuesto permitieron identificar ciertos problemas en el código de detección de obstáculos, que afectaban el desempeño del robot en situaciones de alta demanda. Para solucionar estos problemas, se realizaron ajustes en el código, con el objetivo de mejorar la precisión y eficiencia del sistema. Estos ajustes incluyeron la optimización de los algoritmos de detección de obstáculos, la reducción del tiempo de respuesta del robot y la mejora de la coordinación entre los motores y los sensores.

Los ajustes realizados permitieron al robot moverse de manera más fluida y precisa, evitando los obstáculos de manera oportuna y manteniendo una distancia segura en todo momento. Además, se mejoró la capacidad de respuesta del robot, permitiéndole ajustar su trayectoria de manera rápida y eficiente. Estos ajustes fueron fundamentales para garantizar el correcto funcionamiento del robot en situaciones de alta demanda y para mejorar su desempeño en entornos complejos. Esto se puede ver en el Listing 7.

Listing 7: Tercer ajuste de código

```

1 #include "CYdLidar.h"
2 #include <SFML/Graphics.hpp>
3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>

```

```

12 #include <thread>
13 #include <random>
14 #include <sys/socket.h>
15 #include <sys/un.h>
16 #include <unistd.h>
17 #include <cstring>
18 #include <pthread.h>
19 #include <vector>
20 #include <mutex>
21 #define PI 180.f
22
23 using namespace std;
24 using namespace ydlidar;
25
26 // Pines y configuracin para los motores
27 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los motores
28 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin para los
    motores
29 const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
30 int frequency = 400; // Frecuencia inicial
31
32 std::atomic<bool> is_running(true);
33 std::atomic<bool> is_manual_mode(true);
34
35 void setMotorSpeed(int motor, int frequency) {
36     if (motor >= 0 && motor < 4) {
37         gpioSetPWmfrequency(PWM_PINS[motor], frequency);
38     }
39 }
40
41 void setMotorDirection(int motor, int direction) {
42     if (motor >= 0 && motor < 4) {
43         gpioWrite(DIR_PINS[motor], direction);
44     }
45 }
46
47 void stopMotors() {
48     for (int i = 0; i < 4; ++i) {
49         gpioPWM(PWM_PINS[i], 0);
50     }
51 }
52
53 void moveForward() {
54     for (int i = 0; i < 4; ++i) {
55         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
56     }
57     setMotorDirection(0, 0); // Motor 1
58     setMotorDirection(2, 0); // Motor 3
59     setMotorDirection(1, 1); // Motor 2
60     setMotorDirection(3, 1); // Motor 4
61 }
62
63 void moveBackward() {
64     for (int i = 0; i < 4; ++i) {
65         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
66     }
67     setMotorDirection(0, 1); // Motor 1
68     setMotorDirection(2, 1); // Motor 3
69     setMotorDirection(1, 0); // Motor 2
70     setMotorDirection(3, 0); // Motor 4

```

```

71 }
72
73 void turnLeft() {
74     for (int i = 0; i < 4; ++i) {
75         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
76     }
77     setMotorDirection(0, 1); // Motor 1
78     setMotorDirection(2, 1); // Motor 3
79     setMotorDirection(1, 1); // Motor 2
80     setMotorDirection(3, 1); // Motor 4
81 }
82
83 void turnRight() {
84     for (int i = 0; i < 4; ++i) {
85         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
86     }
87     setMotorDirection(0, 0); // Motor 1
88     setMotorDirection(2, 0); // Motor 3
89     setMotorDirection(1, 0); // Motor 2
90     setMotorDirection(3, 0); // Motor 4
91 }
92
93 void increaseSpeed() {
94     if (frequency < 2000) {
95         frequency += 100;
96         if (frequency > 2000) frequency = 2000;
97         for (int i = 0; i < 4; ++i) {
98             setMotorSpeed(i, frequency);
99         }
100     }
101 }
102
103 void decreaseSpeed() {
104     if (frequency > 400) {
105         frequency -= 100;
106         if (frequency < 400) frequency = 400;
107         for (int i = 0; i < 4; ++i) {
108             setMotorSpeed(i, frequency);
109         }
110     }
111 }
112
113
114 sf::Color getPointColor(float distance, float maxRange) {
115     float ratio = distance / maxRange;
116     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de rojo
117     a verde
118 }
119
120 int contadorObstaculoTotal = 0;
121 int obstaculoHola = 0;
122 int obsRelativo = 0;
123 int idTipoObstaculo = 0;
124 void randomMovement(CYdLidar &laser) {
125     std::random_device rd;
126     std::mt19937 gen(rd());
127     std::discrete_distribution<> dist({0, 5}); // Distribucion para la
128     probabilidad de movimiento
129
130     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15 grados

```

```

    en radaianes
129 const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15 grados
    en radianes
130 const float DETECTION_RADIUS = 0.25f; // 35 cm
131
132 std::vector<float> previousScanPoints;
133
134 while (is_running) {
135     if (!is_manual_mode) {
136         LaserScan scan;
137         if (laser.doProcessSimple(scan)) {
138             bool obstacle_detected = false;
139             std::vector<float> currentScanPoints;
140
141             for (const auto &point : scan.points) {
142                 // Convertir el ngulo del punto al ngulo relativo al "
143                 // sur" del robot
144                 float adjusted_angle = point.angle + PI;
145                 //std::cout << "Er " << point.range << std::endl;
146                 // Verificar si el punto est dentro del rango frontal
147                 // de 30
148                 //std::cout << point.angle << std::endl;
149                 if (point.range > 0 && point.range < 0.30 && ( point.
150                     angle <= -0.5235f && point.angle >= -2.617f)) {
151                     // Norte
152                     obstacle_detected = true;
153                     currentScanPoints.push_back(point.range);
154                     idTipoObstaculo = 1;
155                     std::cout << "Obstacle distance N: " << (float)
156                         point.range << " Y en el angulo " << point.
157                         angle << std::endl;
158                     break;
159                 } else if (point.range > 0 && point.range < 0.25 && (
160                     point.angle <= 0.872f && point.angle >= -0.5235f))
161                 { // Este
162                     obstacle_detected = true;
163                     currentScanPoints.push_back(point.range);
164                     idTipoObstaculo = 2;
165                     std::cout << "Obstacle distance E: " << (float)
166                         point.range << " Y en el angulo " << point.
167                         angle << std::endl;
168                     break;
169                 } else if (point.range > 0 && point.range < 0.45 && (
170                     point.angle <= 2.26f && point.angle >= 0.872f)) {
171                     // Sur
172                     obstacle_detected = true;
173                     currentScanPoints.push_back(point.range);
174                     idTipoObstaculo = 3;
175                     std::cout << "Obstacle distance S:" << (float)
176                         point.range << " Y en el angulo " << point.
177                         angle << std::endl;
178                     break;
179                 } else if (point.range > 0 && point.range < 0.25 && (
180                     point.angle <= -2.61f || point.angle >= 2.27f)) {
181                     // Oeste
182                     obstacle_detected = true;
183                     currentScanPoints.push_back(point.range);
184                     idTipoObstaculo = 4;
185                     std::cout << "Obstacle distance O:" << (float)
186                         point.range << " Y en el angulo " << point.

```

```

170         angle << std::endl;
171         break;
172     }
173
174 }
175
176 if (obstacle_detected) {
177     // Si detecta un obstculo, retrocede por 4 segundos
178     contadorObstaculoTotal++;
179     std::cout << "Obs: " << contadorObstaculoTotal << std
180         ::endl;
181     //obsRelativo++;
182     stopMotors();
183     //Luego gira aleatoriamente a la izquierda o derecha
184     switch(idTipoObstaculo) {
185         case 1: {
186             int turn = dist(gen) % 2;
187             moveBackward();
188             std::this_thread::sleep_for(std::chrono::
189                 seconds(5));
190             if (turn == 0) {
191                 turnLeft();
192                 std::this_thread::sleep_for(std::chrono::
193                     seconds(5));
194             } else {
195                 turnRight();
196                 std::this_thread::sleep_for(std::chrono::
197                     seconds(5));
198             }
199             std::this_thread::sleep_for(std::chrono::
200                 seconds(2));
201             stopMotors();
202         }
203         break;
204
205         case 2:
206             turnLeft();
207             std::this_thread::sleep_for(std::chrono::
208                 seconds(5));
209             std::this_thread::sleep_for(std::chrono::
210                 seconds(2));
211             stopMotors();
212             break;
213
214         case 3:// este es el 4 y el de abajo el 3
215             moveForward();
216             std::this_thread::sleep_for(std::chrono::
217                 seconds(2));
218             stopMotors();
219             break;
220
221         case 4:
222             turnRight();
223             std::this_thread::sleep_for(std::chrono::
224                 seconds(7));
225             stopMotors();
226             break;

```

```

220     default: {
221         int turn = dist(gen) % 2;
222         if (turn == 0) {
223             turnLeft();
224             std::this_thread::sleep_for(std::chrono::
                seconds(5));
225         } else {
226             turnRight();
227             std::this_thread::sleep_for(std::chrono::
                seconds(5));
228         }
229         std::this_thread::sleep_for(std::chrono::
                seconds(2));
230         stopMotors();
231     }
232     break;
233 }
234
235
236 if(!previousScanPoints.empty() && previousScanPoints.
    size() == currentScanPoints.size()){
237     bool mismoObs = true;
238     for(size_t i = 0; i < currentScanPoints.size()
        ; ++i){
239         if(fabs(currentScanPoints[i] -
                previousScanPoints[i]) > 0.05){
240             mismoObs = false;
241             break;
242         }
243     }
244     if(mismoObs){
245         obsRelativo++;
246     }else{
247         obsRelativo = 0;
248     }
249 }
250
251 previousScanPoints = currentScanPoints;
252
253 if((obsRelativo > 3)){
254     //obsRelativo = 0;
255     moveBackward();
256     std::this_thread::sleep_for(std::chrono::seconds
        (3));
257     stopMotors();
258
259
260     switch(idTipoObstaculo) {
261         case 1: {
262             int turn = dist(gen) % 2;
263             moveBackward();
264             std::this_thread::sleep_for(std::chrono::
                seconds(5));
265             if (turn == 0) {
266                 turnLeft();
267                 std::this_thread::sleep_for(std::
                    chrono::seconds(5));
268             } else {
269                 turnRight();
270                 std::this_thread::sleep_for(std::

```

```

271         chrono::seconds(5));
272     }
273     std::this_thread::sleep_for(std::chrono::
274         seconds(2));
275     stopMotors();
276 }
277 break;
278
279 case 2:
280     turnLeft();
281     std::this_thread::sleep_for(std::chrono::
282         seconds(5));
283     std::this_thread::sleep_for(std::chrono::
284         seconds(2));
285     stopMotors();
286 break;
287
288 case 3:
289     turnRight();
290     std::this_thread::sleep_for(std::chrono::
291         seconds(7));
292     stopMotors();
293 break;
294
295 case 4:
296     moveForward();
297     std::this_thread::sleep_for(std::chrono::
298         seconds(2));
299     stopMotors();
300 break;
301
302 default: {
303     int turn = dist(gen) % 2;
304     if (turn == 0) {
305         turnLeft();
306         std::this_thread::sleep_for(std::
307             chrono::seconds(5));
308     } else {
309         turnRight();
310         std::this_thread::sleep_for(std::
311             chrono::seconds(5));
312     }
313     std::this_thread::sleep_for(std::chrono::
314         seconds(2));
315     stopMotors();
316 }
317 break;
318 }
319
320 } else {
321     moveForward();
322     // Si no hay obstculo, elige un movimiento aleatorio
323     //
324     //int move = dist(gen);
325     //switch (move) {
326     // case 0: moveBackward(); break;
327     // case 1: turnRight(); break;

```

```

322             // case 2: turnLeft(); break;
323             // case 3: moveForward(); break;
324         //}
325     }
326 }
327
328     std::this_thread::sleep_for(std::chrono::milliseconds(500));
329     //stopMotors();
330 }
331     std::this_thread::sleep_for(std::chrono::milliseconds(100));
332 }
333 }
334
335 int main() {
336     // Inicializar pigpio
337     if (gpioInitialise() < 0) {
338         std::cerr << "Error: No se pudo inicializar pigpio." << std::endl;
339         return -1;
340     }
341
342     // Configurar pines de direccin como salida
343     for (int i = 0; i < 4; ++i) {
344         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
345         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
346         setMotorSpeed(i, frequency); // Inicializar PWM con frecuencia
            inicial
347     }
348
349     // Asegrate de establecer XDG_RUNTIME_DIR
350     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
351         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
352     }
353
354     // Ejecuta libcamera-vid en un proceso separado y captura la salida en
        YUV, sin previsualizacn
355     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview -o -
        ", "r");
356     if (!pipe) {
357         std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::
            endl;
358         return -1;
359     }
360
361     // Configura la ventana SFML
362     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
        Visualization with LiDAR");
363     sf::Texture cameraTexture;
364     sf::Sprite cameraSprite;
365
366     // Buffer para leer los datos de video
367     const int width = 640;
368     const int height = 480;
369     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
        tamao del buffer para YUV420
370
371     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data());
372     cv::Mat rgbImage(height, width, CV_8UC3);
373
374     std::string port;
375     ydlidar::os_init();

```

```

376
377 // Obtener los puertos disponibles de LiDAR
378 std::map<std::string, std::string> ports = ydlidar::lidarPortList();
379 if (ports.size() > 1) {
380     auto it = ports.begin();
381     std::advance(it, 1); // Selecciona el segundo puerto disponible
382     port = it->second;
383 } else if (ports.size() == 1) {
384     port = ports.begin()->second;
385 } else {
386     std::cerr << "No se detect ningn LiDAR. Verifica la conexin." <<
        std::endl;
387     return -1;
388 }
389
390 // Configuracin del LiDAR
391 int baudrate = 115200;
392 std::cout << "Baudrate: " << baudrate << std::endl;
393
394 CYdLidar laser;
395 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
396 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int));
397
398 bool isSingleChannel = true;
399 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
        bool));
400
401 float max_range = 8.0f;
402 float min_range = 0.1f;
403 float max_angle = 180.0f;
404 float min_angle = -180.0f;
405 float frequency = 8.0f;
406
407 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
408 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
409 laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
410 laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
411 laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float));
412
413 // Inicializar LiDAR
414 if (!laser.initialize()) {
415     std::cerr << "Error al inicializar el LiDAR." << std::endl;
416     return -1;
417 }
418
419 // Iniciar el escaneo
420 if (!laser.turnOn()) {
421     std::cerr << "Error al encender el LiDAR." << std::endl;
422     return -1;
423 }
424
425 // Thread para movimiento aleatorio
426 std::thread randomMoveThread(randomMovement, std::ref(laser));
427
428 while (window.isOpen()) {
429     sf::Event event;
430     while (window.pollEvent(event)) {
431         if (event.type == sf::Event::Closed)
432             window.close();
433

```

```

434     if (event.type == sf::Event::KeyPressed) {
435         switch (event.key.code) {
436             case sf::Keyboard::W:
437                 is_manual_mode = true;
438                 moveForward();
439                 break;
440             case sf::Keyboard::S:
441                 is_manual_mode = true;
442                 moveBackward();
443                 break;
444             case sf::Keyboard::A:
445                 is_manual_mode = true;
446                 turnLeft();
447                 break;
448             case sf::Keyboard::D:
449                 is_manual_mode = true;
450                 turnRight();
451                 break;
452             case sf::Keyboard::V:
453                 is_manual_mode = true;
454                 increaseSpeed();
455                 break;
456             case sf::Keyboard::B:
457                 is_manual_mode = true;
458                 decreaseSpeed();
459                 break;
460             case sf::Keyboard::Space:
461                 is_manual_mode = true;
462                 stopMotors();
463                 break;
464             case sf::Keyboard::K:
465                 is_manual_mode = false;
466                 break;
467             case sf::Keyboard::M:
468                 is_manual_mode = true;
469                 break;
470             default:
471                 break;
472         }
473     }
474 }
475
476 // Leer los datos del video desde la tubera
477 size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe);
478 if (bytesRead != buffer.size()) {
479     std::cerr << "Error: No se pudo leer suficientes datos de
480         video." << std::endl;
481     continue;
482 }
483
484 // Convertir YUV420 a RGB
485 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
486
487 // Convertir a RGBA aadiendo un canal alfa
488 cv::Mat frame_rgba;
489 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
490
491 // Actualizar la textura de la cmara con los datos del frame
492 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
493     std::cerr << "Error: No se pudo crear la textura." << std::

```

```

        endl;
493     continue;
494 }
495 cameraTexture.update(frame_rgba.ptr());
496
497 cameraSprite.setTexture(cameraTexture);
498 cameraSprite.setScale(
499     window.getSize().x / static_cast<float>(cameraTexture.getSize
500     ().x),
501     window.getSize().y / static_cast<float>(cameraTexture.getSize
502     ().y)
503 );
504
505 window.clear();
506 window.draw(cameraSprite);
507
508 // Crear un minimapa para el LiDAR
509 sf::RectangleShape minimap(sf::Vector2f(200, 200));
510 minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
511     semitransparente
512 minimap.setPosition(10, 10); // Esquina superior izquierda
513
514 window.draw(minimap);
515
516 // Dibujar el centro del LiDAR (color azul)
517 sf::CircleShape lidarCenter(5); // Radio del circulo del LiDAR
518 lidarCenter.setFillColor(sf::Color::Blue);
519 lidarCenter.setPosition(105, 105); // Posicin del centro en el
520     minimapa
521
522 window.draw(lidarCenter);
523
524 // Dibujar la lnea hacia el norte
525 sf::Vertex line[] =
526 {
527     sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
528     sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) // Lnea
529     hacia arriba (norte)
530 };
531
532 window.draw(line, 2, sf::Lines);
533
534 LaserScan scan;
535 if (laser.doProcessSimple(scan)) {
536     for (const auto& point : scan.points) {
537         // Convertir coordenadas polares a cartesianas
538         float x = point.range * cos(point.angle);
539         float y = point.range * sin(point.angle);
540
541         // Ajustar los puntos al minimapa
542         float scale = 25.0f;
543         float adjustedX = 110 + x * scale;
544         float adjustedY = 110 - y * scale; // Invertir Y para
545             coordinar con la pantalla
546
547         // Dibujar los puntos en el minimapa, excluyendo el centro
548             (0,0)
549         if (point.range > 0.05) {
550             sf::CircleShape lidarPoint(2);
551             lidarPoint.setPosition(adjustedX, adjustedY);

```

```

545         lidarPoint.setFillColor(getPointColor(point.range,
546             max_range));
547         window.draw(lidarPoint);
548     }
549     //std::cout << "An " << point.range <<std :: endl;
550 }
551 } else {
552     std::cerr << "No se pudieron obtener los datos del LiDAR." <<
553         std::endl;
554 }
555     window.display();
556 }
557
558 // Detener el escaneo del LiDAR
559 laser.turnOff();
560 laser.disconnecting();
561
562 // Cierra la tubera, detiene los motores y apaga el robot
563 pclose(pipe);
564 stopMotors();
565 gpioTerminate();
566
567 is_running = false;
568 randomMoveThread.join();
569
570 return 0;
571 }

```

5.2.5. Ajustes de código en función de pruebas unitarias y de aceptación 3

Durante el desarrollo del sistema, se realizaron pruebas unitarias y de aceptación para verificar el correcto funcionamiento de los módulos y componentes del sistema. Estas pruebas permitieron identificar errores y fallas en el código, así como realizar ajustes y mejoras en el sistema. A continuación, se describen los ajustes realizados en función de las pruebas unitarias y de aceptación.

En el módulo de control de movimiento, se realizaron ajustes para mejorar la precisión y eficiencia del movimiento del robot. Se implementó una función de seguimiento de rutas por línea, que permite al robot seguir una línea dibujada en el minimapa con el mouse. Esta función utiliza los sensores de distancia LIDAR para detectar la línea y ajustar la trayectoria del robot en tiempo real. Además, se optimizó el algoritmo de control de movimiento para reducir el tiempo de respuesta y mejorar la precisión del movimiento.

En el módulo de control de visión, se realizaron ajustes para mejorar la precisión y eficiencia de la detección de obstáculos. Se implementó un algoritmo de detección de obstáculos basado en visión por computadora, que utiliza la cámara de visión nocturna para identificar obstáculos en el entorno del robot. Este algoritmo utiliza técnicas de procesamiento de imágenes para detectar objetos y calcular su distancia y posición relativa al robot. Además, se optimizó el algoritmo de detección de obstáculos para reducir el tiempo de procesamiento y mejorar la precisión de la detección.

Estos ajustes se pueden observar en el código fuente del sistema, que se presenta a continuación en el Listing 8.

Listing 8: Cuarto ajuste de código

```
1 #include "CYdLidar.h"
2 #include <SFML/Graphics.hpp>
3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>
12 #include <thread>
13 #include <random>
14 #include <sys/socket.h>
15 #include <sys/un.h>
16 #include <unistd.h>
17 #include <cstring>
18 #include <pthread.h>
19 #include <vector>
20 #include <mutex>
21 #define PI 180.f
22
23 using namespace std;
24 using namespace ydlidar;
25
26 // Pines y configuracin para los motores
27 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los motores
28 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin para los
    motores
29 const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
30 int frequency = 400; // Frecuencia inicial
31 const sf::Vector2f robotFixedPosition(110, 110);
32 std::atomic<bool> is_running(true);
33 std::atomic<bool> is_manual_mode(true);
34
35 void setMotorSpeed(int motor, int frequency) {
36     if (motor >= 0 && motor < 4) {
37         gpioSetPWmfrequency(PWM_PINS[motor], frequency);
38     }
39 }
40
41 void setMotorDirection(int motor, int direction) {
42     if (motor >= 0 && motor < 4) {
43         gpioWrite(DIR_PINS[motor], direction);
44     }
45 }
46
47 void stopMotors() {
48     for (int i = 0; i < 4; ++i) {
49         gpioPWM(PWM_PINS[i], 0);
50     }
51 }
52
53 void moveForward() {
54     for (int i = 0; i < 4; ++i) {
55         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
56     }
57     setMotorDirection(0, 0); // Motor 1
58     setMotorDirection(2, 0); // Motor 3
```

```

59     setMotorDirection(1, 1); // Motor 2
60     setMotorDirection(3, 1); // Motor 4
61 }
62
63 void moveBackward() {
64     for (int i = 0; i < 4; ++i) {
65         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
66     }
67     setMotorDirection(0, 1); // Motor 1
68     setMotorDirection(2, 1); // Motor 3
69     setMotorDirection(1, 0); // Motor 2
70     setMotorDirection(3, 0); // Motor 4
71 }
72
73 void turnLeft() {
74     for (int i = 0; i < 4; ++i) {
75         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
76     }
77     setMotorDirection(0, 1); // Motor 1
78     setMotorDirection(2, 1); // Motor 3
79     setMotorDirection(1, 1); // Motor 2
80     setMotorDirection(3, 1); // Motor 4
81 }
82
83 void turnRight() {
84     for (int i = 0; i < 4; ++i) {
85         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
86     }
87     setMotorDirection(0, 0); // Motor 1
88     setMotorDirection(2, 0); // Motor 3
89     setMotorDirection(1, 0); // Motor 2
90     setMotorDirection(3, 0); // Motor 4
91 }
92
93 void increaseSpeed() {
94     if (frequency < 2000) {
95         frequency += 100;
96         if (frequency > 2000) frequency = 2000;
97         for (int i = 0; i < 4; ++i) {
98             setMotorSpeed(i, frequency);
99         }
100     }
101 }
102
103 void decreaseSpeed() {
104     if (frequency > 400) {
105         frequency -= 100;
106         if (frequency < 400) frequency = 400;
107         for (int i = 0; i < 4; ++i) {
108             setMotorSpeed(i, frequency);
109         }
110     }
111 }
112
113
114 sf::Color getPointColor(float distance, float maxRange) {
115     float ratio = distance / maxRange;
116     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de rojo
117     a verde

```

```

118
119
120 std::vector<sf::Vector2f> routePoints;
121 bool isDrawing = false; // Bandera para rastrear si el raton esta
    presionado
122
123
124
125
126 void drawLiDARPoints(sf::RenderWindow &window, const LaserScan &
    lidarPoints, const sf::Vector2f &robotPosition, float scale, float
    max_range) {
127     for (const auto& point : lidarPoints.points) {
128         float x = point.range * cos(point.angle);
129         float y = point.range * sin(point.angle);
130
131         float adjustedX = robotFixedPosition.x + (x * scale) -
            robotPosition.x;
132         float adjustedY = robotFixedPosition.y - (y * scale) -
            robotPosition.y; // Invertir Y para la pantalla
133
134         if (point.range > 0.05) { // Excluir puntos cercanos al centro
135             sf::CircleShape lidarPoint(2); // Tamaño del punto
136             lidarPoint.setPosition(adjustedX, adjustedY);
137             lidarPoint.setFillColor(getPointColor(point.range, max_range))
                ; // Color basado en la distancia
138
139             window.draw(lidarPoint);
140         }
141     }
142 }
143
144
145
146
147
148
149 void handleMouseClicked(const sf::Event &event, const sf::RectangleShape &
    minimap) {
150     // Verificar si el raton esta presionado
151     if (event.type == sf::Event::MouseButtonPressed && event.mouseButton.
        button == sf::Mouse::Left) {
152         isDrawing = true; // Marcar que estamos dibujando
153
154         // Verificar si el raton fue soltado
155     } else if (event.type == sf::Event::MouseButtonReleased && event.
        mouseButton.button == sf::Mouse::Left) {
156         isDrawing = false; // Dejar de dibujar
157     }
158 }
159
160 void updateRoute(const sf::RectangleShape &minimap, sf::RenderWindow &
    window) {
161     // Si el raton esta presionado, agregar puntos a la ruta
162     if (isDrawing) {
163         // Obtener la posicion actual del raton
164         sf::Vector2i mousePosition = sf::Mouse::getPosition(window);
165         sf::Vector2f clickPosition(static_cast<float>(mousePosition.x),
            static_cast<float>(mousePosition.y));
166

```

```

167     // Verificar si la posicion del raton esta dentro del minimapa
168     if (minimap.getGlobalBounds().contains(clickPosition)) {
169         sf::Vector2f relativePosition = clickPosition - minimap.
            getPosition();
170         routePoints.push_back(relativePosition);
171         std::cout << "Dibujando punto: (" << relativePosition.x << ",
            " << relativePosition.y << ")" << std::endl;
172     }
173 }
174 }
175
176
177
178 void drawRoute(sf::RenderWindow &window, const std::vector<sf::Vector2f> &
    routePoints, const sf::RectangleShape &minimap) {
179     if (routePoints.size() > 1) {
180         for (size_t i = 0; i < routePoints.size() - 1; ++i) {
181             // Dibujar las l'ineas de la ruta, ajustando las coordenadas
                en funci'on de la posici'on del robot
182             sf::Vertex line[] = {
183                 sf::Vertex(routePoints[i] + minimap.getPosition(), sf::
                    Color::Red), // Ajuste de posici'on para el minimapa
184                 sf::Vertex(routePoints[i + 1] + minimap.getPosition(), sf
                    ::Color::Red)
185             };
186             window.draw(line, 2, sf::Lines);
187         }
188     }
189 }
190
191
192
193
194
195 // Ajustar la referencia para que el norte sea -1.57 rad (sur en
    coordenadas cartesianas es 0 rad)
196 float adjustAngleForNorth(float angle) {
197     float adjustedAngle = angle - (-M_PI / 2); // Ajuste de 90 (norte en
        -90)
198
199     // Asegurarse de que el angulo este dentro de los limites de -PI a PI
200     if (adjustedAngle > M_PI) {
201         adjustedAngle -= 2 * M_PI;
202     } else if (adjustedAngle < -M_PI) {
203         adjustedAngle += 2 * M_PI;
204     }
205
206     return adjustedAngle;
207 }
208
209
210
211
212
213
214
215 void rotateTowardsPoint(float currentAngle, float targetAngle) {
216     float angleDifference = targetAngle - currentAngle;
217
218     // Asegurate de que el angulo este entre -PI y PI

```

```

219     if (angleDifference > M_PI) {
220         angleDifference -= 2 * M_PI;
221     } else if (angleDifference < -M_PI) {
222         angleDifference += 2 * M_PI;
223     }
224
225     // Gira en la direccion correcta
226     if (angleDifference > 0) {
227         turnRight(); // Girar hacia la derecha
228     } else {
229         turnLeft(); // Girar hacia la izquierda
230     }
231
232     // Esperar hasta que el robot este alineado
233     std::this_thread::sleep_for(std::chrono::milliseconds(static_cast<int>
        >(std::fabs(angleDifference) * 1000)));
234
235     stopMotors();
236 }
237
238
239
240
241
242
243 void moveTowardsPoint(float distance) {
244     moveForward(); // Mover el robot hacia adelante
245
246     // Esperar un tiempo proporcional a la distancia
247     std::this_thread::sleep_for(std::chrono::milliseconds(static_cast<int>
        >(distance * 1000)));
248
249     stopMotors(); // Detener el robot
250 }
251
252
253
254
255 void moveTo(sf::Vector2f point, sf::Vector2f &robotPosition, float &
    currentAngle) {
256     // Calcular la diferencia en X e Y
257     float deltaX = point.x - robotPosition.x;
258     float deltaY = robotPosition.y - point.y; // Invertir la Y para la
        pantalla
259
260     // Calcular la distancia al punto
261     float distanceToPoint = sqrt(deltaX * deltaX + deltaY * deltaY);
262
263     // Si la distancia es mayor a un umbral, mover el robot
264     const float threshold = 5.0f; // Tolerancia para considerar que el
        punto ha sido alcanzado
265     if (distanceToPoint > threshold) {
266         // Calcular el angulo hacia el punto de destino
267         float targetAngle = atan2(deltaY, deltaX);
268
269         // Ajustar el angulo para que el norte sea -90°
270         targetAngle = adjustAngleForNorth(targetAngle);
271
272         // Girar hacia el angulo correcto
273         rotateTowardsPoint(currentAngle, targetAngle); // Funcion para

```

```

    girar hacia el \'angulo
274
275 // Mover hacia el punto
276 moveTowardsPoint(distanceToPoint);
277
278 // Actualizar la posicion y el \'angulo del robot
279 robotPosition = point;
280 currentAngle = targetAngle;
281 }
282 }
283
284
285
286 void followRoute(sf::RenderWindow &window, std::vector<sf::Vector2f> &
routePoints, const LaserScan &lidarPoints, const sf::Vector2f &
minimapPosition, float max_range) {
287     sf::Vector2f robotPosition(110, 110); // Posici\'on inicial del robot
en el minimapa
288     float currentAngle = -M_PI / 2; // \'angulo inicial del robot (norte
en -90Â°)
289
290     while (!routePoints.empty()) {
291         sf::Vector2f targetPoint = routePoints.front();
292
293         moveTo(targetPoint, robotPosition, currentAngle);
294
295         routePoints.erase(routePoints.begin());
296
297         drawRoute(window, routePoints, minimapPosition);
298         drawLiDARPoints(window, lidarPoints, robotPosition, 25.0f,
max_range);
299     }
300 }
301
302
303
304
305
306
307
308
309
310
311 int contadorObstaculoTotal = 0;
312 int obstaculoHola = 0;
313 int obsRelativo = 0;
314 int idTipoObstaculo = 0;
315 void randomMovement(CYdLidar &laser) {
316     std::random_device rd;
317     std::mt19937 gen(rd());
318     std::discrete_distribution<> dist({0, 5}); // Distribucion para la
probabilidad de movimiento
319
320     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15 grados
en radaianes
321     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15 grados
en radianes
322     const float DETECTION_RADIUS = 0.25f; // 35 cm
323
324     std::vector<float> previousScanPoints;

```

```

325
326 while (is_running) {
327     if (!is_manual_mode) {
328         LaserScan scan;
329         if (laser.doProcessSimple(scan)) {
330             bool obstacle_detected = false;
331             std::vector<float> currentScanPoints;
332
333             for (const auto &point : scan.points) {
334                 // Convertir el ngulo del punto al ngulo relativo al "
335                 // sur" del robot
336                 float adjusted_angle = point.angle + PI;
337                 //std::cout << "Er " << point.range << std::endl;
338                 // Verificar si el punto est dentro del rango frontal
339                 // de 30
340                 //std::cout << point.angle << std::endl;
341                 if (point.range > 0 && point.range < 0.30 && ( point.
342                     angle <= -0.5235f && point.angle >= -2.617f)) {
343                     // Norte
344                     obstacle_detected = true;
345                     currentScanPoints.push_back(point.range);
346                     idTipoObstaculo = 1;
347                     std::cout << "Obstacle distance N: " << (float)
348                         point.range << " Y en el angulo " << point.
349                         angle << std::endl;
350                     break;
351                 } else if (point.range > 0 && point.range < 0.25 && (
352                     point.angle <= 0.872f && point.angle >= -0.5235f))
353                 { // Este
354                     obstacle_detected = true;
355                     currentScanPoints.push_back(point.range);
356                     idTipoObstaculo = 2;
357                     std::cout << "Obstacle distance E: " << (float)
358                         point.range << " Y en el angulo " << point.
359                         angle << std::endl;
360                     break;
361                 } else if (point.range > 0 && point.range < 0.45 && (
362                     point.angle <= 2.26f && point.angle >= 0.872f))
363                 { // Sur
364                     obstacle_detected = true;
365                     currentScanPoints.push_back(point.range);
366                     idTipoObstaculo = 3;
367                     std::cout << "Obstacle distance S:" << (float)
368                         point.range << " Y en el angulo " << point.
369                         angle << std::endl;
370                     break;
371                 } else if (point.range > 0 && point.range < 0.25 && (
372                     point.angle <= -2.61f || point.angle >= 2.27f))
373                 { // Oeste
374                     obstacle_detected = true;
375                     currentScanPoints.push_back(point.range);
376                     idTipoObstaculo = 4;
377                     std::cout << "Obstacle distance O:" << (float)
378                         point.range << " Y en el angulo " << point.
379                         angle << std::endl;
380                     break;
381                 }
382             }
383         }
384     }
385 }

```

```

367
368     if (obstacle_detected) {
369         // Si detecta un obstculo, retrocede por 4 segundos
370         contadorObstaculoTotal++;
371         std::cout << "Obs: " << contadorObstaculoTotal << std
           ::endl;
372         //obsRelativo++;
373         stopMotors();
374         //Luego gira aleatoriamente a la izquierda o derecha
375         switch(idTipoObstaculo) {
376             case 1: {
377                 int turn = dist(gen) % 2;
378                 moveBackward();
379                 std::this_thread::sleep_for(std::chrono::
           seconds(5));
380                 if (turn == 0) {
381                     turnLeft();
382                     std::this_thread::sleep_for(std::chrono::
           seconds(5));
383                 } else {
384                     turnRight();
385                     std::this_thread::sleep_for(std::chrono::
           seconds(5));
386                 }
387                 std::this_thread::sleep_for(std::chrono::
           seconds(2));
388                 stopMotors();
389             }
390             break;
391
392             case 2:
393                 turnLeft();
394                 std::this_thread::sleep_for(std::chrono::
           seconds(5));
395                 std::this_thread::sleep_for(std::chrono::
           seconds(2));
396                 stopMotors();
397             break;
398
399             case 3:// este es el 4 y el de abajo el 3
400                 moveForward();
401                 std::this_thread::sleep_for(std::chrono::
           seconds(2));
402                 stopMotors();
403
404             break;
405
406             case 4:
407                 turnRight();
408                 std::this_thread::sleep_for(std::chrono::
           seconds(7));
409                 stopMotors();
410             break;
411
412             default: {
413                 int turn = dist(gen) % 2;
414                 if (turn == 0) {
415                     turnLeft();
416                     std::this_thread::sleep_for(std::chrono::
           seconds(5));

```

```

417         } else {
418             turnRight();
419             std::this_thread::sleep_for(std::chrono::
                seconds(5));
420         }
421         std::this_thread::sleep_for(std::chrono::
                seconds(2));
422         stopMotors();
423     }
424     break;
425 }
426
427
428 if(!previousScanPoints.empty() && previousScanPoints.
    size() == currentScanPoints.size()){
429     bool mismoObs = true;
430     for(size_t i = 0; i < currentScanPoints.size()
        ; ++i){
431         if(fabs(currentScanPoints[i] -
                previousScanPoints[i]) > 0.05){
432             mismoObs = false;
433             break;
434         }
435     }
436     if(mismoObs){
437         obsRelativo++;
438     }else{
439         obsRelativo = 0;
440     }
441 }
442
443 previousScanPoints = currentScanPoints;
444
445 if((obsRelativo > 3)){
446     //obsRelativo = 0;
447     moveBackward();
448     std::this_thread::sleep_for(std::chrono::seconds
        (3));
449     stopMotors();
450
451
452     switch(idTipoObstaculo) {
453         case 1: {
454             int turn = dist(gen) % 2;
455             moveBackward();
456             std::this_thread::sleep_for(std::chrono::
                seconds(5));
457             if (turn == 0) {
458                 turnLeft();
459                 std::this_thread::sleep_for(std::
                    chrono::seconds(5));
460             } else {
461                 turnRight();
462                 std::this_thread::sleep_for(std::
                    chrono::seconds(5));
463             }
464             std::this_thread::sleep_for(std::chrono::
                seconds(2));
465             stopMotors();
466         }

```

```

467         break;
468
469         case 2:
470             turnLeft();
471             std::this_thread::sleep_for(std::chrono::
472                 seconds(5));
473             std::this_thread::sleep_for(std::chrono::
474                 seconds(2));
475             stopMotors();
476         break;
477
478         case 3:
479             turnRight();
480             std::this_thread::sleep_for(std::chrono::
481                 seconds(7));
482             stopMotors();
483         break;
484
485         case 4:
486             moveForward();
487             std::this_thread::sleep_for(std::chrono::
488                 seconds(2));
489             stopMotors();
490         break;
491
492         default: {
493             int turn = dist(gen) % 2;
494             if (turn == 0) {
495                 turnLeft();
496                 std::this_thread::sleep_for(std::
497                     chrono::seconds(5));
498             } else {
499                 turnRight();
500                 std::this_thread::sleep_for(std::
501                     chrono::seconds(5));
502             }
503             std::this_thread::sleep_for(std::chrono::
504                 seconds(2));
505             stopMotors();
506         }
507         break;
508     }
509 }
510
511 } else {
512     moveForward();
513     // Si no hay obstculo, elige un movimiento aleatorio
514     //
515     //int move = dist(gen);
516     //switch (move) {
517     //    // case 0: moveBackward(); break;
518     //    // case 1: turnRight(); break;
519     //    // case 2: turnLeft(); break;
520     //    // case 3: moveForward(); break;
521     //}
522 }
523 }

```

```

520         std::this_thread::sleep_for(std::chrono::milliseconds(500));
521         //stopMotors();
522     }
523     std::this_thread::sleep_for(std::chrono::milliseconds(100));
524 }
525 }
526
527
528
529
530
531
532 int main() {
533     // Inicializar pigpio
534     if (gpioInitialise() < 0) {
535         std::cerr << "Error: No se pudo inicializar pigpio." << std::endl;
536         return -1;
537     }
538
539     // Configurar pines de direccin como salida
540     for (int i = 0; i < 4; ++i) {
541         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
542         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
543         setMotorSpeed(i, frequency); // Inicializar PWM con frecuencia
544         inicial
545     }
546
547     // Asegrate de establecer XDG_RUNTIME_DIR
548     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
549         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
550     }
551
552     // Ejecuta libcamera-vid en un proceso separado y captura la salida en
553     // YUV, sin previsualizacn
554     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview -o -
555     ", "r");
556     if (!pipe) {
557         std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::
558         endl;
559         return -1;
560     }
561
562     // Configura la ventana SFML
563     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
564     Visualization with LiDAR");
565     sf::Texture cameraTexture;
566     sf::Sprite cameraSprite;
567
568     // Buffer para leer los datos de video
569     const int width = 640;
570     const int height = 480;
571     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
572     tamao del buffer para YUV420
573
574     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data());
575     cv::Mat rgbImage(height, width, CV_8UC3);
576
577     std::string port;
578     ydlidar::os_init();
579

```

```

574 // Obtener los puertos disponibles de LiDAR
575 std::map<std::string, std::string> ports = ydlidar::lidarPortList();
576 if (ports.size() > 1) {
577     auto it = ports.begin();
578     std::advance(it, 1); // Selecciona el segundo puerto disponible
579     port = it->second;
580 } else if (ports.size() == 1) {
581     port = ports.begin()->second;
582 } else {
583     std::cerr << "No se detect ningn LiDAR. Verifica la conexin." <<
584         std::endl;
585     return -1;
586 }
587 // Configuracin del LiDAR
588 int baudrate = 115200;
589 std::cout << "Baudrate: " << baudrate << std::endl;
590
591 CYdLidar laser;
592 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
593 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int));
594
595 bool isSingleChannel = true;
596 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
597     bool));
598
599 float max_range = 8.0f;
600 float min_range = 0.1f;
601 float max_angle = 180.0f;
602 float min_angle = -180.0f;
603 float frequency = 8.0f;
604
605 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
606 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
607 laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
608 laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
609 laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float));
610
611 // Inicializar LiDAR
612 if (!laser.initialize()) {
613     std::cerr << "Error al inicializar el LiDAR." << std::endl;
614     return -1;
615 }
616
617 // Iniciar el escaneo
618 if (!laser.turnOn()) {
619     std::cerr << "Error al encender el LiDAR." << std::endl;
620     return -1;
621 }
622
623 // Thread para movimiento aleatorio
624 std::thread randomMoveThread(randomMovement, std::ref(laser));
625
626 while (window.isOpen()) {
627     // Crear un minimapa para el LiDAR
628     sf::RectangleShape minimap(sf::Vector2f(200, 200));
629     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
630         semitransparente
631     minimap.setPosition(10, 10); // Esquina superior izquierda

```

```

631
632 // Leer los datos del video desde la tubera
633 size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe);
634 if (bytesRead != buffer.size()) {
635     std::cerr << "Error: No se pudo leer suficientes datos de
636         video." << std::endl;
637     continue;
638 }
639 // Convertir YUV420 a RGB
640 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
641
642 // Convertir a RGBA aadiendo un canal alfa
643 cv::Mat frame_rgba;
644 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
645
646 // Actualizar la textura de la cmara con los datos del frame
647 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
648     std::cerr << "Error: No se pudo crear la textura." << std:::
649         endl;
650     continue;
651 }
652 cameraTexture.update(frame_rgba.ptr());
653
654 cameraSprite.setTexture(cameraTexture);
655 cameraSprite.setScale(
656     window.getSize().x / static_cast<float>(cameraTexture.getSize
657         ().x),
658     window.getSize().y / static_cast<float>(cameraTexture.getSize
659         ().y)
660 );
661
662 // Actualizar la ruta en el minimapa
663
664 window.clear();
665 window.draw(cameraSprite);
666
667 window.draw(minimap);
668 updateRoute(minimap, window);
669 // Dibujar la ruta en el minimapa
670 drawRoute(window, routePoints, minimap);
671 // Dibujar el centro del LiDAR (color azul)
672 sf::CircleShape lidarCenter(5); // Radio del crculo del LiDAR
673 lidarCenter.setFillColor(sf::Color::Blue);
674 lidarCenter.setPosition(105, 105); // Posicin del centro en el
675     minimapa
676
677 window.draw(lidarCenter);
678
679 // Dibujar la lnea hacia el norte
680 sf::Vertex line[] =
681 {
682     sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
683     sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) // Lnea
684     hacia arriba (norte)
685 };
686
687 window.draw(line, 2, sf::Lines);

```

```

685
686 LaserScan scan;
687 if (laser.doProcessSimple(scan)) {
688     drawLiDARPoints(window, scan, minimap.getPosition(), 25.0f,
689                     max_range);
690 }else {
691     std::cerr << "No se pudieron obtener los datos del LiDAR." <<
692     std::endl;
693 }
694
695 sf::Event event;
696 while (window.pollEvent(event)) {
697     if (event.type == sf::Event::Closed){
698         window.close();
699     }
700
701     if(event.type == sf::Event::MouseButtonPressed){
702         std::cout << "Mouse pressed" << std::endl;
703         handleMouseClick(event, minimap);
704     }else if(event.type == sf::Event::MouseButtonReleased){
705         std::cout << "Mouse released" << std::endl;
706         handleMouseClick(event, minimap);
707         isDrawing = false;
708     }
709
710 }
711
712 if (event.type == sf::Event::KeyPressed) {
713     switch (event.key.code) {
714         case sf::Keyboard::W:
715             is_manual_mode = true;
716             std::cout << "W" << std::endl;
717             moveForward();
718             break;
719         case sf::Keyboard::S:
720             is_manual_mode = true;
721             std::cout << "S" << std::endl;
722             moveBackward();
723             break;
724         case sf::Keyboard::A:
725             is_manual_mode = true;
726             std::cout << "A" << std::endl;
727             turnLeft();
728             break;
729         case sf::Keyboard::D:
730             is_manual_mode = true;
731             std::cout << "D" << std::endl;
732             turnRight();
733             break;
734         case sf::Keyboard::V:
735             is_manual_mode = true;
736             std::cout << "V" << std::endl;
737             increaseSpeed();
738             break;
739         case sf::Keyboard::B:
740             is_manual_mode = true;
741             std::cout << "B" << std::endl;
742             decreaseSpeed();
743             break;

```

```

743         case sf::Keyboard::Space:
744             is_manual_mode = true;
745             std::cout << "Space" << std::endl;
746             stopMotors();
747             break;
748         case sf::Keyboard::R: {
749             is_manual_mode = false;
750             std::thread routeThread(followRoute, std::ref(
                window), std::ref(routePoints), std::ref(scan)
                , minimap.getPosition(), max_range);
                routeThread.detach();
751             break;
752         }
753     case sf::Keyboard::K:
754         is_manual_mode = false;
755         std::cout << "K" << std::endl;
756         break;
757     case sf::Keyboard::M:
758         is_manual_mode = true;
759         std::cout << "M" << std::endl;
760         break;
761     default:
762         break;
763     }
764 }
765 }
766 }
767
768
769     window.display();
770 }
771
772 // Detener el escaneo del LiDAR
773 laser.turnOff();
774 laser.disconnecting();
775
776 // Cierra la tubera, detiene los motores y apaga el robot
777 pclose(pipe);
778 stopMotors();
779 gpioTerminate();
780
781 is_running = false;
782 randomMoveThread.join();
783
784 return 0;
785 }

```

Después de eso, se mejoro o se cambio mas bien la función de seguimiento de rutas por medio de un valor numérico, que se obtiene del algoritmo del Physarum, se puede ver la función en el Listing 9.

Listing 9: Sexto ajuste de código

```

1     void moveAlongRoute(const std::vector<int> &route) {
2         // Las celdas est\'an numeradas de 1 a N como en la imagen
           proporcionada.
3         std::map<int, std::pair<int, int>> cellCoordinates = {
4             {1, {0, 0}}, {2, {0, 1}}, {3, {0, 2}},
5             {4, {1, 0}}, {5, {1, 1}}, {6, {1, 2}},
6             {7, {2, 0}}, {8, {2, 1}}, {9, {2, 2}},
7         // Contin\'ua con el resto de las celdas de la cuadr\'
           icula...

```

```

8         {10,{3, 0}}, {11,{3, 1}}, {12,{3, 2}},
9         {13,{4, 0}}, {14,{4, 1}}, {15,{4, 2}},
10        {16,{5, 0}}, {17,{5, 1}}, {18,{5, 2}},
11        {19,{6, 0}}, {20,{6, 1}}, {21,{6, 2}},
12        {22,{7, 0}}, {23,{7, 1}}, {24,{7, 2}}
13    };
14
15    std::cout << "Moviendo a lo largo de la ruta..." << std::endl;
16    std::cout << "La ruta es: " << std::endl;
17    for (int cell : route) {
18        std::cout << cell << " ";
19    }
20    std::cout << std::endl;
21    for (size_t i = 0; i < route.size() - 1; ++i) {
22        int currentCell = route[i];
23        int nextCell = route[i + 1];
24        std::cout << "Current cell: " << currentCell << " Next
25            cell: " << nextCell << endl;
26        auto currentCoord = cellCoordinates[currentCell];
27        auto nextCoord = cellCoordinates[nextCell];
28
29        // Calculamos la direcci\on en la que el robot debe
30        moverse.
31        int deltaX = nextCoord.first - currentCoord.first;
32        int deltaY = nextCoord.second - currentCoord.second;
33        is_manual_mode = false;
34        std::cout << "Delta X: " << deltaX << " Delta Y: " <<
35            deltaY << endl;
36        if (deltaX == 0 && deltaY == 1) {
37            // Movimiento hacia la derecha.
38            stopMotors();
39            std::cout << "Girando hacia la derecha y avanzando..."
40                << std::endl;
41            turnRight();
42            std::this_thread::sleep_for(std::chrono::milliseconds
43                (4500)); // Girar por 4.35 segundos.
44            stopMotors();
45            std::this_thread::sleep_for(std::chrono::milliseconds
46                (500)); // Pausa antes de avanzar.
47            moveForward();
48            std::this_thread::sleep_for(std::chrono::milliseconds
49                (5000)); // Avanzar por 5 segundos.
50            stopMotors();
51        } else if (deltaX == 0 && deltaY == -1) {
52            // Movimiento hacia la izquierda.
53            stopMotors();
54            std::cout << "Girando hacia la izquierda y avanzando
55                ..." << std::endl;
56            turnLeft();
57            std::this_thread::sleep_for(std::chrono::milliseconds
58                (4350)); // Girar por 4.35 segundos.
59            stopMotors();
60            std::this_thread::sleep_for(std::chrono::milliseconds
61                (500)); // Pausa antes de avanzar.
62            moveForward();
63            std::this_thread::sleep_for(std::chrono::milliseconds
64                (5000)); // Avanzar por 5 segundos.
65            stopMotors();
66        } else if (deltaX == 1 && deltaY == 0) {
67            // Movimiento hacia adelante (eje X positivo).

```

```

57         stopMotors();
58         std::cout << "Avanzando hacia adelante..." << std::
           endl;
59         moveForward();
60         std::this_thread::sleep_for(std::chrono::milliseconds
           (5000)); // Avanzar por 5 segundos.
61         stopMotors();
62     } else if (deltaX == -1 && deltaY == 0) {
63         // Movimiento hacia atr\'as (eje X negativo).
64         std::cout << "Retrocediendo..." << std::endl;
65         moveBackward();
66         std::this_thread::sleep_for(std::chrono::milliseconds
           (5000)); // Retroceder por 5 segundos.
67         stopMotors();
68     } else {
69         std::cerr << "Movimiento no soportado entre las celdas
           : " << currentCell << " y " << nextCell << std::
           endl;
70     }
71     std::cout << "Movimiento completado." << std::endl;
72     stopMotors();
73     // Pausa entre movimientos para evitar sobrecargar los
       motores.
74     std::this_thread::sleep_for(std::chrono::milliseconds
       (1000));
75     }
76 }

```

5.2.6. Ajustes de código en función de las pruebas de carga 2

Luego de realizar las pruebas de carga 2, se identificaron algunos problemas en el código del robot, específicamente en el algoritmo de seguimiento de ruta. Se observó que el robot tenía dificultades para seguir la ruta de manera precisa, lo que resultaba en movimientos erráticos y desviaciones significativas de la trayectoria deseada. Para solucionar este problema, se realizaron los siguientes ajustes en el código del robot:

- Se mejoró el algoritmo de seguimiento de ruta para que el robot pudiera seguir la trayectoria de manera más precisa. Se ajustaron los parámetros de control de velocidad y dirección para minimizar las desviaciones y los errores en el seguimiento de la ruta.
- Se añadió un nuevo apartado gráfico en la interfaz de usuario para mostrar las dimensiones del robot en 2D, lo que permitió al operador visualizar la posición y orientación del robot en relación con la ruta planificada. Esto facilitó la supervisión y el control del robot durante la operación, y ayudó a prevenir colisiones y desviaciones no deseadas.

Estos ajustes permitieron mejorar el desempeño del robot y garantizar un seguimiento preciso de la ruta, lo que resultó en un funcionamiento más eficiente y fiable del sistema en general. Estos cambios fueron fundamentales para optimizar el rendimiento del robot y garantizar su correcto funcionamiento durante las pruebas de carga y las operaciones en entornos reales, y se pueden observar en el Listing 10.

Listing 10: Quinto ajuste de código

```

1     #include "CYdLidar.h"
2     #include <SFML/Graphics.hpp>

```

```

3     #include <opencv2/opencv.hpp>
4     #include <iostream>
5     #include <cstdlib>
6     #include <cstdio>
7     #include <pigpio.h>
8     #include <string>
9     #include <map>
10    #include <vector>
11    #include <atomic>
12    #include <thread>
13    #include <random>
14    #include <sys/socket.h>
15    #include <sys/un.h>
16    #include <unistd.h>
17    #include <cstring>
18    #include <pthread.h>
19    #include <vector>
20    #include <mutex>
21    #define PI 180.f
22
23    using namespace std;
24    using namespace ydlidar;
25
26    // Pines y configuracin para los motores
27    const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
        motores
28    const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin
        para los motores
29    const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
30    int frequency = 400; // Frecuencia inicial
31    const sf::Vector2f robotFixedPosition(110, 110);
32    std::atomic<bool> is_running(true);
33    std::atomic<bool> is_manual_mode(true);
34
35    void setMotorSpeed(int motor, int frequency) {
36        if (motor >= 0 && motor < 4) {
37            gpioSetPWmfrequency(PWM_PINS[motor], frequency);
38        }
39    }
40
41    void setMotorDirection(int motor, int direction) {
42        if (motor >= 0 && motor < 4) {
43            gpioWrite(DIR_PINS[motor], direction);
44        }
45    }
46
47    void stopMotors() {
48        for (int i = 0; i < 4; ++i) {
49            gpioPWM(PWM_PINS[i], 0);
50        }
51    }
52
53    void moveForward() {
54        for (int i = 0; i < 4; ++i) {
55            gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
                al 50%
56        }
57        setMotorDirection(0, 0); // Motor 1
58        setMotorDirection(2, 0); // Motor 3
59        setMotorDirection(1, 1); // Motor 2

```

```

60         setMotorDirection(3, 1); // Motor 4
61     }
62
63     void moveBackward() {
64         for (int i = 0; i < 4; ++i) {
65             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
66                 al 50%
67         }
68         setMotorDirection(0, 1); // Motor 1
69         setMotorDirection(2, 1); // Motor 3
70         setMotorDirection(1, 0); // Motor 2
71         setMotorDirection(3, 0); // Motor 4
72     }
73
74     void turnLeft() {
75         std::cout << "Turning left" << std::endl;
76         for (int i = 0; i < 4; ++i) {
77             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
78                 al 50%
79         }
80         setMotorDirection(0, 1); // Motor 1
81         setMotorDirection(2, 1); // Motor 3
82         setMotorDirection(1, 1); // Motor 2
83         setMotorDirection(3, 1); // Motor 4
84     }
85
86     void turnRight() {
87         std::cout << "Turning right" << std::endl;
88         for (int i = 0; i < 4; ++i) {
89             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
90                 al 50%
91         }
92         setMotorDirection(0, 0); // Motor 1
93         setMotorDirection(2, 0); // Motor 3
94         setMotorDirection(1, 0); // Motor 2
95         setMotorDirection(3, 0); // Motor 4
96     }
97
98     void increaseSpeed() {
99         if (frequency < 2000) {
100             frequency += 100;
101             if (frequency > 2000) frequency = 2000;
102             for (int i = 0; i < 4; ++i) {
103                 setMotorSpeed(i, frequency);
104             }
105         }
106     }
107
108     void decreaseSpeed() {
109         if (frequency > 400) {
110             frequency -= 100;
111             if (frequency < 400) frequency = 400;
112             for (int i = 0; i < 4; ++i) {
113                 setMotorSpeed(i, frequency);
114             }
115         }
116     }
117
118     sf::Color getPointColor(float distance, float maxRange) {

```

```

117         float ratio = distance / maxRange;
118         return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
           de rojo a verde
119     }
120
121
122     std::vector<sf::Vector2f> routePoints;
123     bool isDrawing = false; // Bandera para rastrear si el raton esta
           presionado
124
125
126
127
128     void drawLiDARPoints(sf::RenderWindow &window, const LaserScan &
           lidarPoints, const sf::Vector2f &robotPosition, float scale,
           float max_range) {
129         for (const auto& point : lidarPoints.points) {
130             float x = point.range * cos(point.angle);
131             float y = point.range * sin(point.angle);
132
133             float adjustedX = robotFixedPosition.x + (x * scale) -
           robotPosition.x;
134             float adjustedY = robotFixedPosition.y - (y * scale) -
           robotPosition.y; // Invertir Y para la pantalla
135
136             if (point.range > 0.05) { // Excluir puntos cercanos al
           centro
137                 sf::CircleShape lidarPoint(2); // Tamao del punto
138                 lidarPoint.setPosition(adjustedX, adjustedY);
139                 lidarPoint.setFillColor(getPointColor(point.range,
           max_range)); // Color basado en la distancia
140
141                 window.draw(lidarPoint);
142             }
143         }
144     }
145
146
147
148
149
150
151     void handleClick(const sf::Event &event, const sf::
           RectangleShape &minimap) {
152         // Verificar si el raton esta presionado
153         if (event.type == sf::Event::MouseButtonPressed && event.
           mouseButton.button == sf::Mouse::Left) {
154             isDrawing = true; // Marcar que estamos dibujando
155
156             // Verificar si el raton fue soltado
157         } else if (event.type == sf::Event::MouseButtonReleased &&
           event.mouseButton.button == sf::Mouse::Left) {
158             isDrawing = false; // Dejar de dibujar
159         }
160     }
161
162     void updateRoute(const sf::RectangleShape &minimap, sf::
           RenderWindow &window) {
163         // Si el raton esta presionado, agregar puntos a la ruta
164         if (isDrawing) {

```

```

165         // Obtener la posicion actual del raton
166         sf::Vector2i mousePosition = sf::Mouse::getPosition(window
167         );
168         sf::Vector2f clickPosition(static_cast<float>(
169         mousePosition.x), static_cast<float>(mousePosition.y))
170         ;
171
172         // Verificar si la posicion del raton esta dentro del
173         minimapa
174         if (minimap.getGlobalBounds().contains(clickPosition)) {
175             sf::Vector2f relativePosition = clickPosition -
176             minimap.getPosition();
177             routePoints.push_back(relativePosition);
178             std::cout << "Dibujando punto: (" << relativePosition.
179             x << ", " << relativePosition.y << ")" << std::
180             endl;
181         }
182     }
183 }
184
185 void drawRoute(sf::RenderWindow &window, const std::vector<sf::
186 Vector2f> &routePoints, const sf::Vector2f &robotPosition) {
187     if (routePoints.size() > 1) {
188         for (size_t i = 0; i < routePoints.size() - 1; ++i) {
189             // Dibujar las lineas de la ruta, ajustando las
190             coordenadas en funcin de la posicin del robot
191             sf::Vertex line[] = {
192                 sf::Vertex(robotFixedPosition + (routePoints[i] -
193                 robotPosition), sf::Color::Red), // Ajustar
194                 posicin en funcin del robot
195                 sf::Vertex(robotFixedPosition + (routePoints[i +
196                 1] - robotPosition), sf::Color::Red)
197             };
198             window.draw(line, 2, sf::Lines);
199         }
200     }
201 }
202
203 // Ajustar la referencia para que el norte sea -1.57 rad (sur en
204 coordenadas cartesianas es 0 rad)
205 float adjustAngleForNorth(float angle) {
206     float adjustedAngle = angle - (-M_PI / 2); // Ajuste de 90 (
207     norte en -90)
208
209     // Asegurarse de que el angulo este dentro de los limites de -
210     PI a PI
211     if (adjustedAngle > M_PI) {
212         adjustedAngle -= 2 * M_PI;
213     } else if (adjustedAngle < -M_PI) {
214         adjustedAngle += 2 * M_PI;
215     }
216
217     return adjustedAngle;
218 }

```

```

210
211
212
213
214
215
216
217 void rotateTowardsPoint(float currentAngle, float targetAngle) {
218     float angleDifference = targetAngle - currentAngle;
219
220     // Asegurate de que el angulo este entre -PI y PI
221     if (angleDifference > M_PI) {
222         angleDifference -= 2 * M_PI;
223     } else if (angleDifference < -M_PI) {
224         angleDifference += 2 * M_PI;
225     }
226
227     // Gira en la direccion correcta
228     if (angleDifference > 0) {
229         turnRight(); // Girar hacia la derecha
230     } else {
231         turnLeft(); // Girar hacia la izquierda
232     }
233
234     // Esperar hasta que el robot este alineado
235     std::this_thread::sleep_for(std::chrono::milliseconds(
        static_cast<int>(std::fabs(angleDifference) * 1000)));
236
237     stopMotors();
238 }
239
240
241
242
243
244
245 void moveTowardsPoint(float distance) {
246     moveForward(); // Mover el robot hacia adelante
247
248     // Esperar un tiempo proporcional a la distancia
249     std::this_thread::sleep_for(std::chrono::milliseconds(
        static_cast<int>(distance * 1000)));
250
251     stopMotors(); // Detener el robot
252 }
253
254
255
256
257 void moveTo(sf::Vector2f point, sf::Vector2f &robotPosition, float
    &currentAngle) {
258     // Calcular la diferencia en X e Y
259     float deltaX = point.x - robotPosition.x;
260     float deltaY = robotPosition.y - point.y; // Invertir la Y
        para la pantalla
261
262     // Calcular la distancia al punto
263     float distanceToPoint = sqrt(deltaX * deltaX + deltaY * deltaY
        );
264

```

```

265     // Si la distancia es mayor a un umbral, mover el robot
266     const float threshold = 5.0f; // Tolerancia para considerar
        que el punto ha sido alcanzado
267     if (distanceToPoint > threshold) {
268         // Calcular el angulo hacia el punto de destino
269         float targetAngle = atan2(deltaY, deltaX);
270         // Ajustar el angulo para que el norte sea -90
271         targetAngle = adjustAngleForNorth(targetAngle);
272
273         // Girar hacia el angulo correcto
274         rotateTowardsPoint(currentAngle, targetAngle); // Funcin
        para girar hacia el ngulo
275
276         // Mover hacia el punto
277         moveTowardsPoint(distanceToPoint);
278
279         // Actualizar la posicion y el ngulo del robot
280         robotPosition = point;
281         currentAngle = targetAngle;
282     }
283 }
284
285
286
287 void followRoute(sf::RenderWindow &window, std::vector<sf::
    Vector2f> &routePoints, const LaserScan &lidarPoints, const sf
    ::Vector2f &minimapPosition, float max_range) {
288     sf::Vector2f robotPosition(110, 110); // Posicin inicial del
        robot en el minimapa
289     float currentAngle = -M_PI / 2; // ngulo inicial del robot (
        norte en -90)
290
291     while (!routePoints.empty()) {
292         sf::Vector2f targetPoint = routePoints.front();
293
294         moveTo(targetPoint, robotPosition, currentAngle);
295
296         routePoints.erase(routePoints.begin());
297
298         drawRoute(window, routePoints, minimapPosition);
299         drawLiDARPoints(window, lidarPoints, robotPosition, 25.0f,
            max_range);
300     }
301 }
302
303
304
305
306
307
308
309
310
311
312 int contadorObstaculoTotal = 0;
313 int obstaculoHola = 0;
314 int obsRelativo = 0;
315 int idTipoObstaculo = 0;
316 void randomMovement(CYdLidar &laser) {
317     std::random_device rd;

```

```

318     std::mt19937 gen(rd());
319     std::discrete_distribution<> dist({0, 5}); // Distribucion para
        la probabilidad de movimiento
320
321     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
        grados en radiaianes
322     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
        grados en radianes
323     const float DETECTION_RADIUS = 0.25f; // 35 cm
324
325     std::vector<float> previousScanPoints;
326
327     while (is_running) {
328         if (!is_manual_mode) {
329             LaserScan scan;
330             if (laser.doProcessSimple(scan)) {
331                 bool obstacle_detected = false;
332                 std::vector<float> currentScanPoints;
333
334                 for (const auto &point : scan.points) {
335                     // Convertir el ngulo del punto al ngulo
                        relativo al "sur" del robot
336                     float adjusted_angle = point.angle + PI;
337                     //std::cout << "Er " << point.range << std::
                        endl;
338                     // Verificar si el punto est dentro del rango
                        frontal de 30
339                     //std::cout << point.angle << std::endl;
340                     if (point.range > 0 && point.range < 0.30 && (
                        point.angle <= -0.5235f && point.angle
                        >= -2.617f)) { // Norte
341                         obstacle_detected = true;
342                         currentScanPoints.push_back(point.range);
343                         idTipoObstaculo = 1;
344                         std::cout << "Obstacle distance N: " << (
                            float)point.range << " Y en el angulo
                            "<<point.angle << std::endl;
345                         break;
346                     } else if (point.range > 0 && point.range <
                        0.25 && (point.angle <= 0.872f && point.
                        angle >= -0.5235f)){ // Este
347                         obstacle_detected = true;
348                         currentScanPoints.push_back(point.range);
349                         idTipoObstaculo = 2;
350                         std::cout << "Obstacle distance E: " << (
                            float)point.range << " Y en el angulo
                            "<<point.angle << std::endl;
351                         break;
352                     }else if (point.range > 0 && point.range < 0.45
                        && (point.angle <= 2.26f && point.angle
                        >= 0.872f)){ // Sur
353                         obstacle_detected = true;
354                         currentScanPoints.push_back(point.range);
355                         idTipoObstaculo = 3;
356                         std::cout << "Obstacle distance S:" << (
                            float)point.range << " Y en el angulo
                            "<<point.angle << std::endl;
357                         break;
358                     }else if (point.range > 0 && point.range <
                        0.25 && (point.angle <= -2.61f || point.

```

```

359         angle >= 2.27f)){ // Oeste
360         obstacle_detected = true;
361         currentScanPoints.push_back(point.range);
362         idTipoObstaculo = 4;
363         std::cout << "Obstacle distance 0:" << (
364             float)point.range << " Y en el angulo
365             "<<point.angle << std::endl;
366         break;
367     }
368 }
369
370 if (obstacle_detected) {
371     // Si detecta un obstculo, retrocede por 4
372     segundos
373     contadorObstaculoTotal++;
374     std::cout << "Obs: " << contadorObstaculoTotal
375     << std::endl;
376     //obsRelativo++;
377     stopMotors();
378     //Luego gira aleatoriamente a la izquierda o
379     derecha
380     switch(idTipoObstaculo) {
381         case 1: {
382             int turn = dist(gen) % 2;
383             moveBackward();
384             std::this_thread::sleep_for(std::
385                 chrono::seconds(5));
386             if (turn == 0) {
387                 turnLeft();
388                 std::this_thread::sleep_for(std::
389                     chrono::seconds(5));
390             } else {
391                 turnRight();
392                 std::this_thread::sleep_for(std::
393                     chrono::seconds(5));
394             }
395             std::this_thread::sleep_for(std::
396                 chrono::seconds(2));
397             stopMotors();
398         }
399         break;
400
401         case 2:
402             turnLeft();
403             std::this_thread::sleep_for(std::
404                 chrono::seconds(5));
405             std::this_thread::sleep_for(std::
406                 chrono::seconds(2));
407             stopMotors();
408         break;
409
410         case 3:// este es el 4 y el de abajo el 3
411             moveForward();
412             std::this_thread::sleep_for(std::
413                 chrono::seconds(2));
414             stopMotors();
415         break;

```

```

406
407         case 4:
408             turnRight();
409             std::this_thread::sleep_for(std::
410                 chrono::seconds(7));
411             stopMotors();
412         break;
413     default: {
414         int turn = dist(gen) % 2;
415         if (turn == 0) {
416             turnLeft();
417             std::this_thread::sleep_for(std::
418                 chrono::seconds(5));
419         } else {
420             turnRight();
421             std::this_thread::sleep_for(std::
422                 chrono::seconds(5));
423         }
424         std::this_thread::sleep_for(std::
425             chrono::seconds(2));
426         stopMotors();
427     }
428     break;
429 }
430
431 if(!previousScanPoints.empty() &&
432     previousScanPoints.size() ==
433     currentScanPoints.size()){
434     bool mismoObs = true;
435     for(size_t i = 0; i <
436         currentScanPoints.size(); ++i){
437         if(fabs(currentScanPoints[i] -
438             previousScanPoints[i]) > 0.05)
439         {
440             mismoObs = false;
441             break;
442         }
443     }
444     if(mismoObs){
445         obsRelativo++;
446     }else{
447         obsRelativo = 0;
448     }
449 }
450
451 previousScanPoints = currentScanPoints;
452
453 if((obsRelativo > 3)){
454     //obsRelativo = 0;
455     moveBackward();
456     std::this_thread::sleep_for(std::chrono::
457         seconds(3));
458     stopMotors();
459
460     switch(idTipoObstaculo) {
461         case 1: {
462             int turn = dist(gen) % 2;

```

```

456         moveBackward();
457         std::this_thread::sleep_for(std::
            chrono::seconds(5));
458     if (turn == 0) {
459         turnLeft();
460         std::this_thread::sleep_for(
            std::chrono::seconds(5));
461     } else {
462         turnRight();
463         std::this_thread::sleep_for(
            std::chrono::seconds(5));
464     }
465     std::this_thread::sleep_for(std::
        chrono::seconds(2));
466     stopMotors();
467 }
468 break;
469
470 case 2:
471     turnLeft();
472     std::this_thread::sleep_for(std::
        chrono::seconds(5));
473     std::this_thread::sleep_for(std::
        chrono::seconds(2));
474     stopMotors();
475 break;
476
477 case 3:
478     turnRight();
479     std::this_thread::sleep_for(std::
        chrono::seconds(7));
480     stopMotors();
481 break;
482
483 case 4:
484     moveForward();
485     std::this_thread::sleep_for(std::
        chrono::seconds(2));
486     stopMotors();
487 break;
488
489 default: {
490     int turn = dist(gen) % 2;
491     if (turn == 0) {
492         turnLeft();
493         std::this_thread::sleep_for(
            std::chrono::seconds(5));
494     } else {
495         turnRight();
496         std::this_thread::sleep_for(
            std::chrono::seconds(5));
497     }
498     std::this_thread::sleep_for(std::
        chrono::seconds(2));
499     stopMotors();
500 }
501 break;
502 }
503
504

```

```

505     }
506   } else {
507     moveForward();
508     // Si no hay obstaculo, elige un movimiento
509     // aleatorio
510     //
511     //int move = dist(gen);
512     //switch (move) {
513     //  case 0: moveBackward(); break;
514     //  case 1: turnRight(); break;
515     //  case 2: turnLeft(); break;
516     //  case 3: moveForward(); break;
517     //}
518   }
519 }
520
521     std::this_thread::sleep_for(std::chrono::milliseconds
522     (500));
523     //stopMotors();
524 }
525     std::this_thread::sleep_for(std::chrono::milliseconds(100)
526     );
527 }
528
529 // Definir las dimensiones del robot en el minimapa
530 const float frontDistance = 15.0f; // cm hacia adelante
531 const float backDistance = 35.0f; // cm hacia atrs
532 const float rightDistance = 21.0f; // cm hacia la derecha
533 const float leftDistance = 21.0f; // cm hacia la izquierda
534
535 // Escalar las distancias al minimapa (considera ajustar el
536 // factor de escala segn lo necesites)
537 float scale = 0.4f; // Cambia este valor si necesitas ajustar
538 // el tamao del robot en el minimapa
539
540 void drawRobot(sf::RenderWindow &window, const sf::Vector2f &
541 robotPosition, const sf::RectangleShape &minimap) {
542
543     float width = (leftDistance + rightDistance) * scale;
544     float height = (frontDistance + backDistance) * scale;
545
546     // Dibujar el rectngulo representando al robot
547     sf::RectangleShape robotShape(sf::Vector2f(width, height));
548     robotShape.setFillColor(sf::Color(100, 100, 255, 150)); //
549     // Color azul claro con algo de transparencia
550
551     // Ajustar el origen del rectngulo al punto donde est el LiDAR
552     // , es decir, al centro del robot
553     robotShape.setOrigin(leftDistance * scale, backDistance *
554     scale);
555
556     // Establecer la posicin del rectngulo del robot en el
557     // minimapa, centrado en la posicin del LiDAR
558     robotShape.setPosition(105, 105); // Usamos la misma posicin
559     // del LiDAR para centrar el robot

```

```

554         window.draw(robotShape);
555     }
556 }
557
558
559
560
561 void moveAlongRoute(const std::vector<int> &route) {
562     // Las celdas est\`an numeradas de 1 a N como en la imagen
563     // proporcionada.
564     std::map<int, std::pair<int, int>> cellCoordinates = {
565         {1, {0, 0}}, {2, {0, 1}}, {3, {0, 2}},
566         {4, {1, 0}}, {5, {1, 1}}, {6, {1, 2}},
567         {7, {2, 0}}, {8, {2, 1}}, {9, {2, 2}},
568         // Contin\`ua con el resto de las celdas de la cuadr\`
569         // icula...
570         {10,{3, 0}}, {11,{3, 1}}, {12,{3, 2}},
571         {13,{4, 0}}, {14,{4, 1}}, {15,{4, 2}},
572         {16,{5, 0}}, {17,{5, 1}}, {18,{5, 2}},
573         {19,{6, 0}}, {20,{6, 1}}, {21,{6, 2}},
574         {22,{7, 0}}, {23,{7, 1}}, {24,{7, 2}}
575     };
576
577     std::cout << "Moviendo a lo largo de la ruta..." << std::endl;
578     std::cout << "La ruta es: " << std::endl;
579     for (int cell : route) {
580         std::cout << cell << " ";
581     }
582     std::cout << std::endl;
583     for (size_t i = 0; i < route.size() - 1; ++i) {
584         int currentCell = route[i];
585         int nextCell = route[i + 1];
586         std::cout << "Current cell: " << currentCell << " Next
587             cell: " << nextCell << endl;
588         auto currentCoord = cellCoordinates[currentCell];
589         auto nextCoord = cellCoordinates[nextCell];
590
591         // Calculamos la direcci\`on en la que el robot debe
592         // moverse.
593         int deltaX = nextCoord.first - currentCoord.first;
594         int deltaY = nextCoord.second - currentCoord.second;
595         is_manual_mode = false;
596         std::cout << "Delta X: " << deltaX << " Delta Y: " <<
597             deltaY << endl;
598         if (deltaX == 0 && deltaY == 1) {
599             // Movimiento hacia la derecha.
600             stopMotors();
601             std::cout << "Girando hacia la derecha y avanzando..."
602                 << std::endl;
603             turnRight();
604             std::this_thread::sleep_for(std::chrono::milliseconds
605                 (4500)); // Girar por 4.35 segundos.
606             stopMotors();
607             std::this_thread::sleep_for(std::chrono::milliseconds
608                 (500)); // Pausa antes de avanzar.
609             moveForward();
610             std::this_thread::sleep_for(std::chrono::milliseconds
611                 (5000)); // Avanzar por 5 segundos.
612             stopMotors();
613         } else if (deltaX == 0 && deltaY == -1) {

```

```

605         // Movimiento hacia la izquierda.
606         stopMotors();
607         std::cout << "Girando hacia la izquierda y avanzando
        ..." << std::endl;
608         turnLeft();
609         std::this_thread::sleep_for(std::chrono::milliseconds
        (4350)); // Girar por 4.35 segundos.
610         stopMotors();
611         std::this_thread::sleep_for(std::chrono::milliseconds
        (500)); // Pausa antes de avanzar.
612         moveForward();
613         std::this_thread::sleep_for(std::chrono::milliseconds
        (5000)); // Avanzar por 5 segundos.
614         stopMotors();
615     } else if (deltaX == 1 && deltaY == 0) {
616         // Movimiento hacia adelante (eje X positivo).
617         stopMotors();
618         std::cout << "Avanzando hacia adelante..." << std::
        endl;
619         moveForward();
620         std::this_thread::sleep_for(std::chrono::milliseconds
        (5000)); // Avanzar por 5 segundos.
621         stopMotors();
622     } else if (deltaX == -1 && deltaY == 0) {
623         // Movimiento hacia atr\'as (eje X negativo).
624         std::cout << "Retrocediendo..." << std::endl;
625         moveBackward();
626         std::this_thread::sleep_for(std::chrono::milliseconds
        (5000)); // Retroceder por 5 segundos.
627         stopMotors();
628     } else {
629         std::cerr << "Movimiento no soportado entre las celdas
        : " << currentCell << " y " << nextCell << std::
        endl;
630     }
631     std::cout << "Movimiento completado." << std::endl;
632     stopMotors();
633     // Pausa entre movimientos para evitar sobrecargar los
        motores.
634     std::this_thread::sleep_for(std::chrono::milliseconds
        (1000));
635 }
636
637
638
639
640
641
642
643
644
645
646 void drawGrid(sf::RenderWindow &window, const sf::RectangleShape &
        minimap) {
647     const float cellSizeCm = 50.0f; // Tama\~no de cada celda en
        cent\'imetros
648     const float scaleFactor = 10.0f; // Relaci\'on 1 cm = 10 p\'
        ixeles en el minimapa
649     const float cellSizePx = cellSizeCm * scaleFactor; // Tama\~
        no de cada celda en p\'ixeles

```

```

650
651 // Obtener la posici\on y tama\~no del minimapa
652 sf::Vector2f minimapPosition = minimap.getPosition();
653 sf::Vector2f minimapSize = minimap.getSize();
654
655 // Calcular el origen del grid para mantenerlo fijo respecto
656 // al robot centrado en (105, 105)
657 float originX = minimapPosition.x - fmod(minimapPosition.x +
658     105.0f, cellSizePx);
659 float originY = minimapPosition.y - fmod(minimapPosition.y +
660     105.0f, cellSizePx);
661
662 // Dibujar las l\ineas verticales de la cuadr\icula
663 for (float x = originX; x <= minimapPosition.x + minimapSize.x
664     ; x += cellSizePx) {
665     sf::Vertex verticalLine[] = {
666         sf::Vertex(sf::Vector2f(x, minimapPosition.y), sf::
667             Color(150, 150, 150, 200)),
668         sf::Vertex(sf::Vector2f(x, minimapPosition.y +
669             minimapSize.y), sf::Color(150, 150, 150, 200))
670     };
671     window.draw(verticalLine, 2, sf::Lines);
672 }
673
674 // Dibujar las l\ineas horizontales de la cuadr\icula
675 for (float y = originY; y <= minimapPosition.y + minimapSize.y
676     ; y += cellSizePx) {
677     sf::Vertex horizontalLine[] = {
678         sf::Vertex(sf::Vector2f(minimapPosition.x, y), sf::
679             Color(150, 150, 150, 200)),
680         sf::Vertex(sf::Vector2f(minimapPosition.x +
681             minimapSize.x, y), sf::Color(150, 150, 150, 200))
682     };
683     window.draw(horizontalLine, 2, sf::Lines);
684 }
685
686 }
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

699     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
700         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
701     }
702
703     // Ejecuta libcamera-vid en un proceso separado y captura la
704     // salida en YUV, sin previsualizacn
705     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
706     nopreview -o -, "r");
707     if (!pipe) {
708         std::cerr << "Error: No se pudo ejecutar libcamera-vid."
709         << std::endl;
710         return -1;
711     }
712
713     // Configura la ventana SFML
714     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
715     Visualization with LiDAR");
716     sf::Texture cameraTexture;
717     sf::Sprite cameraSprite;
718
719     // Buffer para leer los datos de video
720     const int width = 640;
721     const int height = 480;
722     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
723     // el tamao del buffer para YUV420
724
725     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
726     data());
727     cv::Mat rgbImage(height, width, CV_8UC3);
728
729     std::string port;
730     ydlidar::os_init();
731
732     // Obtener los puertos disponibles de LiDAR
733     std::map<std::string, std::string> ports = ydlidar::
734     lidarPortList();
735     if (ports.size() > 1) {
736         auto it = ports.begin();
737         std::advance(it, 1); // Selecciona el segundo puerto
738         // disponible
739         port = it->second;
740     } else if (ports.size() == 1) {
741         port = ports.begin()->second;
742     } else {
743         std::cerr << "No se detect ningn LiDAR. Verifica la
744         conexin." << std::endl;
745         return -1;
746     }
747
748     // Configuracin del LiDAR
749     int baudrate = 115200;
750     std::cout << "Baudrate: " << baudrate << std::endl;
751
752     CYdLidar laser;
753     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
754     ());
755     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
756     int));
757
758     bool isSingleChannel = true;

```

```

748     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
749                       sizeof(bool));
750
751     float max_range = 8.0f;
752     float min_range = 0.1f;
753     float max_angle = 180.0f;
754     float min_angle = -180.0f;
755     float frequency = 8.0f;
756
757     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
758                       );
759     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
760                       );
761     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
762                       );
763     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
764                       );
765     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
766                       float));
767
768     // Inicializar LiDAR
769     if (!laser.initialize()) {
770         std::cerr << "Error al inicializar el LiDAR." << std::endl
771                 ;
772         return -1;
773     }
774
775     // Iniciar el escaneo
776     if (!laser.turnOn()) {
777         std::cerr << "Error al encender el LiDAR." << std::endl;
778         return -1;
779     }
780
781     // Thread para movimiento aleatorio
782     std::thread randomMoveThread(randomMovement, std::ref(laser));
783
784     while (window.isOpen()) {
785         // Crear un minimapa para el LiDAR
786         sf::RectangleShape minimap(sf::Vector2f(200, 200));
787         minimap.setFillColor(sf::Color(200, 200, 200, 150)); //
788             Fondo semitransparente
789         minimap.setPosition(10, 10); // Esquina superior izquierda
790
791         // Leer los datos del video desde la tubera
792         size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
793             pipe);
794         if (bytesRead != buffer.size()) {
795             std::cerr << "Error: No se pudo leer suficientes datos
796                 de video." << std::endl;
797             continue;
798         }
799
800         // Convertir YUV420 a RGB
801         cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
802
803         // Convertir a RGBA aadiendo un canal alfa
804         cv::Mat frame_rgba;
805         cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
806
807

```

```

798 // Actualizar la textura de la cmara con los datos del
799 // frame
800 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows
801 )) {
802     std::cerr << "Error: No se pudo crear la textura." <<
803     std::endl;
804     continue;
805 }
806 cameraTexture.update(frame_rgba.ptr());
807
808 cameraSprite.setTexture(cameraTexture);
809 cameraSprite.setScale(
810     window.getSize().x / static_cast<float>(cameraTexture.
811     getSize().x),
812     window.getSize().y / static_cast<float>(cameraTexture.
813     getSize().y)
814 );
815
816 // Actualizar la ruta en el minimapa
817
818 window.clear();
819 window.draw(cameraSprite);
820
821 window.draw(minimap);
822 drawGrid(window, minimap);
823 updateRoute(minimap, window);
824 // Dibujar la ruta en el minimapa
825 drawRoute(window, routePoints, minimap.getPosition());
826 // Dibujar el centro del LiDAR (color azul)
827 drawRobot(window, robotFixedPosition, minimap);
828
829 // Calcular el centro del rectngulo del robot para dibujar
830 // la linea que indica el norte
831 sf::Vector2f robotCenter = minimap.getPosition() +
832     robotFixedPosition;
833
834 // Dibujar la linea hacia el norte (ajustada al centro del
835 // rectngulo)
836 sf::Vertex line[] =
837 {
838     sf::Vertex(sf::Vector2f(105, 105), sf::Color::Black),
839     // Posicin inicial de la linea
840     sf::Vertex(sf::Vector2f(105, 160), sf::Color::Black)
841     // Posicin final hacia arriba
842 };
843
844 // Dibujar la linea que representa el norte
845 window.draw(line, 2, sf::Lines);
846
847 LaserScan scan;
848 if (laser.doProcessSimple(scan)) {
849     drawLiDARPoints(window, scan, minimap.getPosition(),
850         25.0f, max_range);
851 }else {
852     std::cerr << "No se pudieron obtener los datos del

```

```

847         LiDAR." << std::endl;
848     }
849     sf::Event event;
850     while (window.pollEvent(event)) {
851         if (event.type == sf::Event::Closed){
852             window.close();
853         }
854
855         if(event.type == sf::Event::MouseButtonPressed){
856             std::cout << "Mouse pressed" << std::endl;
857             handleMouseClicked(event, minimap);
858         }else if(event.type == sf::Event::MouseButtonReleased)
859         {
860             std::cout << "Mouse released" << std::endl;
861             handleMouseClicked(event, minimap);
862             isDrawing = false;
863         }
864
865
866
867         if (event.type == sf::Event::KeyPressed) {
868             switch (event.key.code) {
869                 case sf::Keyboard::W:
870                     is_manual_mode = true;
871                     std::cout << "W" << std::endl;
872                     moveForward();
873                     break;
874                 case sf::Keyboard::S:
875                     is_manual_mode = true;
876                     std::cout << "S" << std::endl;
877                     moveBackward();
878                     break;
879                 case sf::Keyboard::A:
880                     is_manual_mode = true;
881                     std::cout << "A" << std::endl;
882                     turnLeft();
883                     break;
884                 case sf::Keyboard::D:
885                     is_manual_mode = true;
886                     std::cout << "D" << std::endl;
887                     turnRight();
888                     break;
889                 case sf::Keyboard::V:
890                     is_manual_mode = true;
891                     std::cout << "V" << std::endl;
892                     increaseSpeed();
893                     break;
894                 case sf::Keyboard::B:
895                     is_manual_mode = true;
896                     std::cout << "B" << std::endl;
897                     decreaseSpeed();
898                     break;
899                 case sf::Keyboard::Space:
900                     is_manual_mode = true;
901                     std::cout << "Space" << std::endl;
902                     stopMotors();
903                     break;
904                 case sf::Keyboard::R: {

```

```

905         is_manual_mode = false;
906         std::vector<int> route = {1, 2, 5, 8, 11,
907             14, 13, 16};
908         moveAlongRoute(route);
909         //std::thread routeThread(followRoute, std
910             ::ref(window), std::ref(routePoints),
911             std::ref(scan), minimap.getPosition(),
912             max_range);
913         //routeThread.detach();
914         break;
915     }
916     case sf::Keyboard::K:
917         is_manual_mode = false;
918         std::cout << "K" << std::endl;
919         break;
920     case sf::Keyboard::M:
921         is_manual_mode = true;
922         std::cout << "M" << std::endl;
923         break;
924     default:
925         break;
926 }
927 }
928
929     window.display();
930 }
931
932 // Detener el escaneo del LiDAR
933 laser.turnOff();
934 laser.disconnecting();
935
936 // Cierra la tubera, detiene los motores y apaga el robot
937 pclose(pipe);
938 stopMotors();
939 gpioTerminate();
940
941 is_running = false;
942 randomMoveThread.join();
943
944 return 0;
945 }

```

6. Pruebas del sistema

En esta sección se describen y documentan las diferentes pruebas realizadas al sistema desarrollado. Estas pruebas se realizaron con el objetivo de verificar el correcto funcionamiento del sistema y de identificar posibles errores o fallas en el código. Las pruebas se realizaron en diferentes etapas del desarrollo del sistema, desde la implementación de los módulos hasta la integración de los mismos. Primero para cada uno de los productos desarrollados, el simulador del Physarum Polycephalum y el robot, se describen como son las pruebas que se realizaron y los resultados obtenidos. Posteriormente se describen las pruebas correspondientes para cada uno de los productos.

6.1. Pruebas unitarias del algoritmo en simulación de mapas 1

Para la realización de un programa, es necesario realizar las pruebas unitarias correspondientes para comprobar el funcionamiento de cada una de las partes involucradas con el programa.

En la realización de las pruebas unitarias, se utilizó Google Test, el cual es un framework de Google que funciona para el lenguaje de programación C++, que es el lenguaje utilizado para el desarrollo del simulador del Physarum. Es importante recordar que las pruebas unitarias son las que se encargan de verificar que cada una de las partes que componen a un software funcionen bien una a una, por lo que el mayor uso que se le dió al framework es la realización de las pruebas unitarias al código del simulador del Physarum, verificando cada una de las funciones que lo componen y demás componentes del software.

Principalmente, fue probado la implementación del algoritmo, el cual contiene algunas funciones importantes y algunas otras que devuelven valores importantes para la correcta implementación del algoritmo.

Algunas de las pruebas unitarias que fueron aplicadas se pueden ver en la Figura 45.

```
hello_test.cc | TEST(phy_functions, SomeFunctions)
1  #include <gtest/gtest.h>
2  #include "LoadMap.hpp"
3  #include "App.h"
4  #include "Physarum.hpp"
5  LoadMap loadmap;
6  Physarum phy_app;
7  App app;
8  // Demonstrate some basic assertions.
9  TEST(app_functions, SomeFunctions) {
10  EXPECT_EQ(0, app.run()); //Test OK
11  EXPECT_EQ(0, app.processEvents()); //Test OK
12  EXPECT_EQ(0, app.update(sf::Time)); //Test OK
13  EXPECT_EQ(0, app.render()); //Test OK
14  EXPECT_EQ(0, app.handlePlayerInput(sf::Keyboard::Key, bool)); //Test OK
15  EXPECT_EQ(0, app.handleMouseEvents(sf::Mouse::Button, bool)); //Test OK
16  EXPECT_EQ(0, app.setPhysarumOnTexture()); //Test OK
17  EXPECT_EQ(0, app.setTextSettings()); //Test OK
18  EXPECT_EQ(0, app.updateText()); //Test OK
19  EXPECT_EQ(0, app.setGeneralFont()); //Test OK
20  EXPECT_EQ(0, app.initializeColors()); //Test OK
21
22 }
23 // Demonstrate some basic assertions.
24 TEST(phy_functions, SomeFunctions) {
25  EXPECT_EQ(0, phy_app.allNutrientsFounded()); //Test OK
26  EXPECT_EQ(0, phy_app.setCellState()); //Test OK
27  EXPECT_EQ(0, phy_app.cleanCells(sf::Time)); //Test OK
28  EXPECT_EQ(0, phy_app.showPhysarum()); //Test OK
29  EXPECT_EQ(0, phy_app.evaluatePhysarum()); //Test OK
30  EXPECT_EQ(0, phy_app.physarumTransitionConditions()); //Test OK
31  EXPECT_EQ(0, phy_app.getRoute()); //Test OK
32 }
```

Figura 45: Prueba unitaria 1

Como se puede ver, gran parte de las pruebas realizadas al algoritmo fueron satisfactorias a este punto, por lo que podemos ver que a pesar de que la implementación fue en su mayoría correcta, hay una pequeña ventana de oportunidad para la mejora y corrección de la implementación del algoritmo en determinadas circunstancias, donde a pesar de que casi toda la implementación no está relacionada a una entrada del usuario, sino a partes involucradas en la realización de la simulación, entonces hay muy pocos casos en los que una parte de este mismo falle, ya que, debido a la implementación de las reglas, la manera en la que funciona un autómata celular y del propio funcionamiento del algoritmo en sí, no es permitido que algo falle ya que esto alteraría completamente el resultado final del algoritmo, realizando cálculos erróneos en la simulación y errando el objetivo del algoritmo.

6.2. Evaluación de desempeño del software en pruebas iniciales 1

El apartado de desempeño del software es complicado de medir, debido a que el único caso en el que se puede observar si hay una mejora o no es con el tiempo en el cual se realiza la

simulación.

Este tiempo puede variar en cada una de las ejecuciones del programa sin importar si las condiciones iniciales son las mismas, debido a que existe un grado de aleatoriedad inherente al algoritmo, por lo que cada una de las rutas que pueden llegar a ser obtenidas por el simulador varían en cada una de las ejecuciones, como se puede comprobar al observar las Figuras 46 - 50.

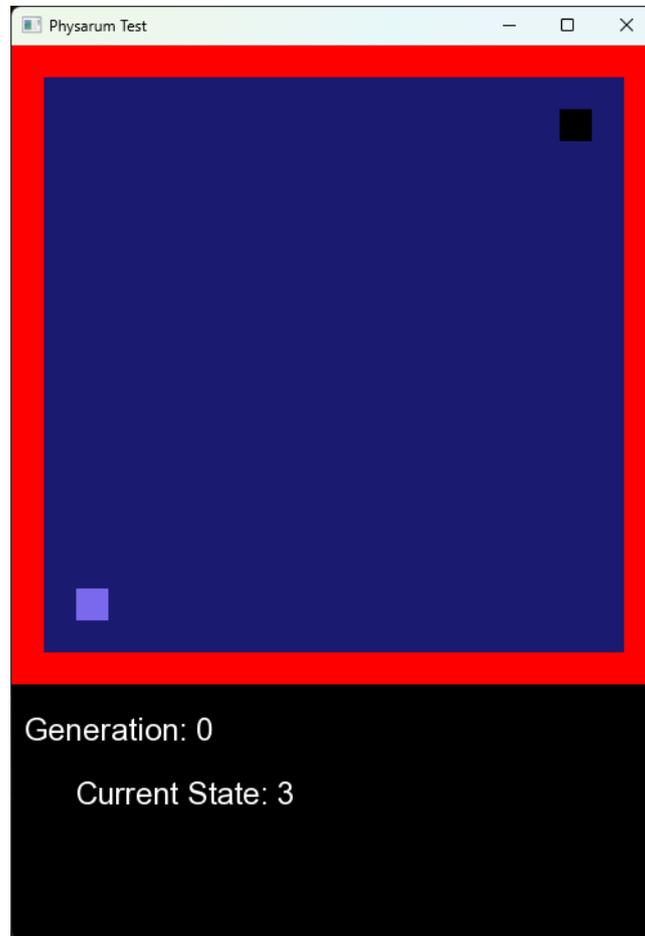


Figura 46: Configuración Base

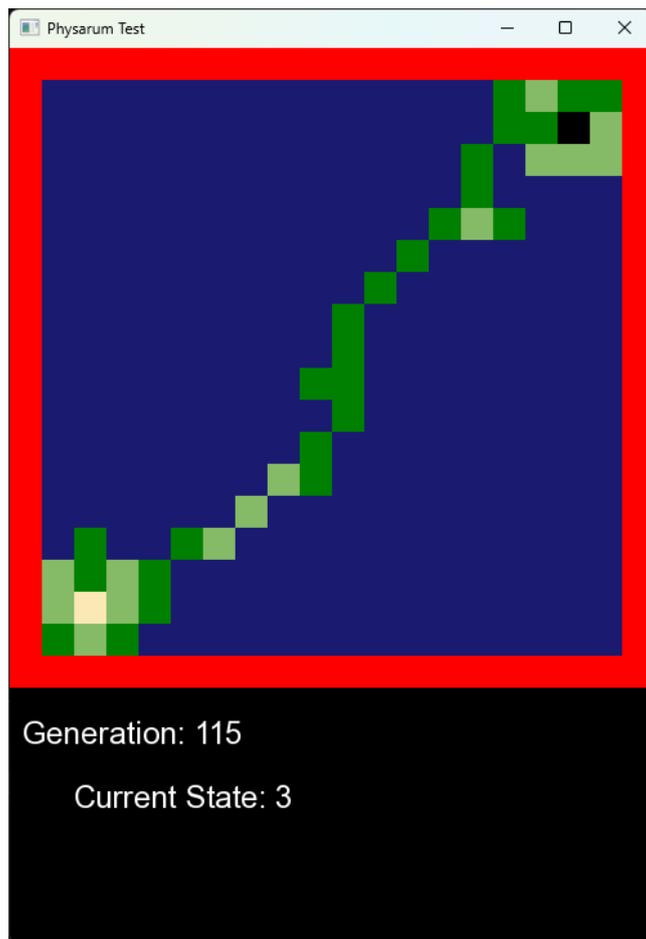


Figura 47: Primera Ruta Obtenida

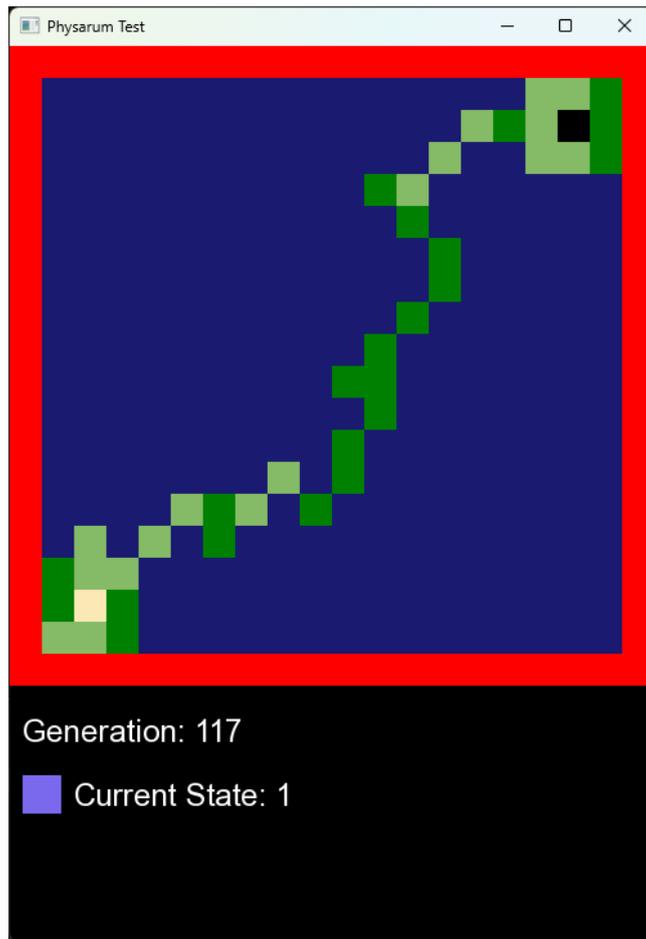


Figura 48: Segunda Ruta Obtenida

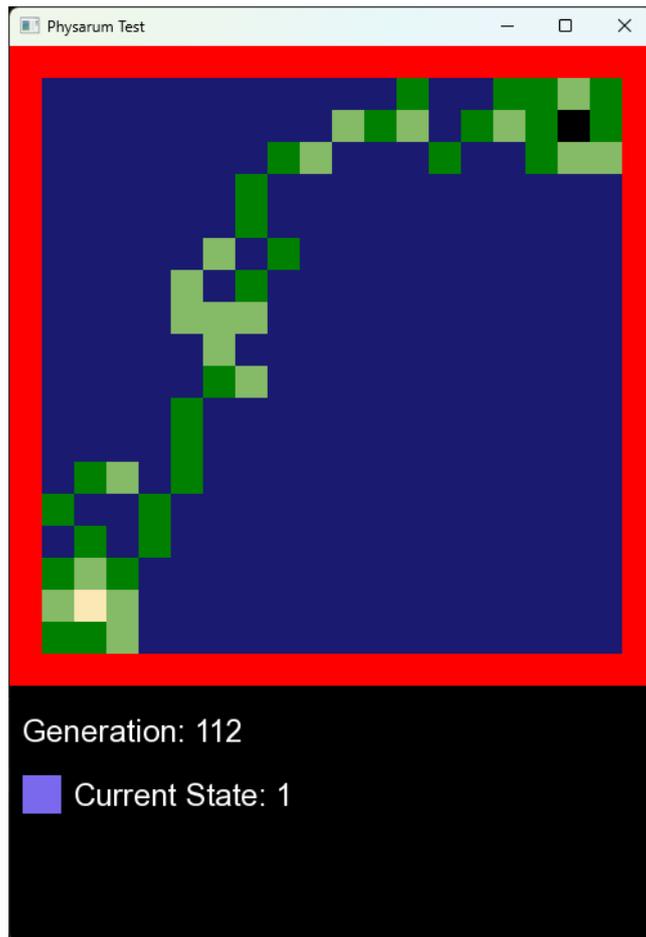


Figura 49: Tercera Ruta Obtenida

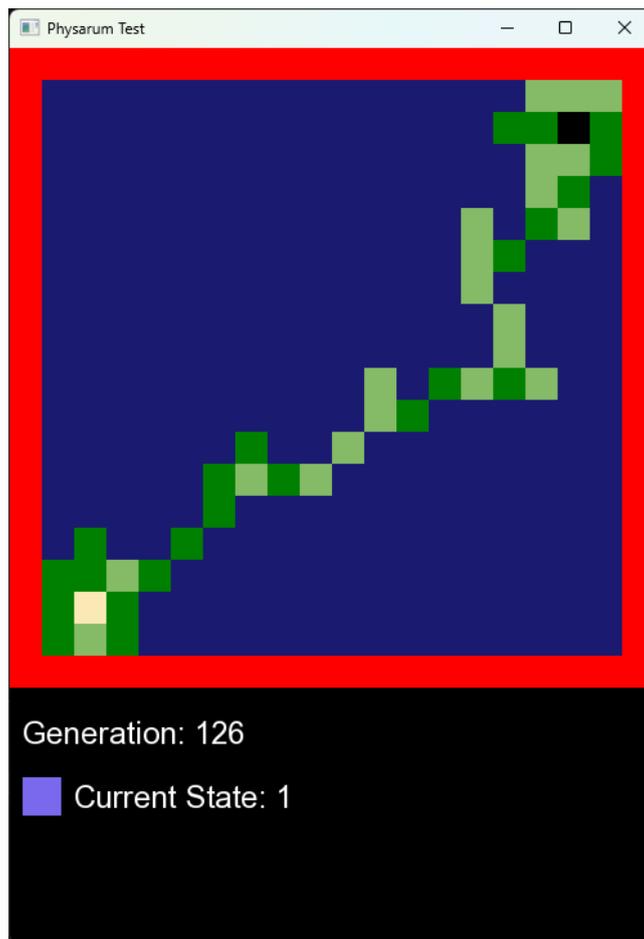


Figura 50: Cuarta Ruta Obtenida

Como se puede comprobar, a pesar de que el estado inicial sea exactamente el mismo, la ruta obtenida a partir del simulador cambia en cada uno de los casos, por lo que medir el tiempo en el que se es obtenida la ruta se vuelve inviable, ya que cada una de las rutas que puede ser generada puede llegar a tardar mayor o menor tiempo de acuerdo a las generaciones que sean necesarias para terminar por obtenerla.

Debido a lo anterior, se consideró otra forma de medir el tiempo de ejecución del programa para analizar su desempeño, es evaluar el tiempo de ejecución entre una generación y la siguiente. Al ser evaluada una matriz de tamaño $n \times n$, donde en cada una se evalúan una serie de condiciones para determinar el valor que contiene dicha celda, hay una diferencia de tiempo entre cada evaluación de la matriz mientras pasa cada una de las generaciones, por lo que se vuelve conveniente usar esta diferencia de tiempo como un parámetro de evaluación del desempeño para el algoritmo.

Durante las pruebas de funcionamiento iniciales del programa, también se encontró que existe una diferencia de desempeño en su ejecución entre Windows y Linux, por lo que para la recopilación de información y datos recabados se debe de tomar en cuenta las ejecuciones en ambos sistemas operativos, debido a que pueden existir diferencias significativas a pesar de que las condiciones iniciales sean similares entre sí.

6.3. Recopilación de datos del software y desempeño 1

La recopilación de datos sobre el software y desempeño fueron basadas en distintas configuraciones iniciales, las cuales fueron introducidas en el simulador como la condición inicial de la simulación. Estas pruebas fueron, además, recopiladas tanto en el sistema operativo de Windows como el de Linux, que son las dos plataformas en las cuales el programa puede funcionar. Los datos recopilados se muestran de las Figuras 51 - 61.

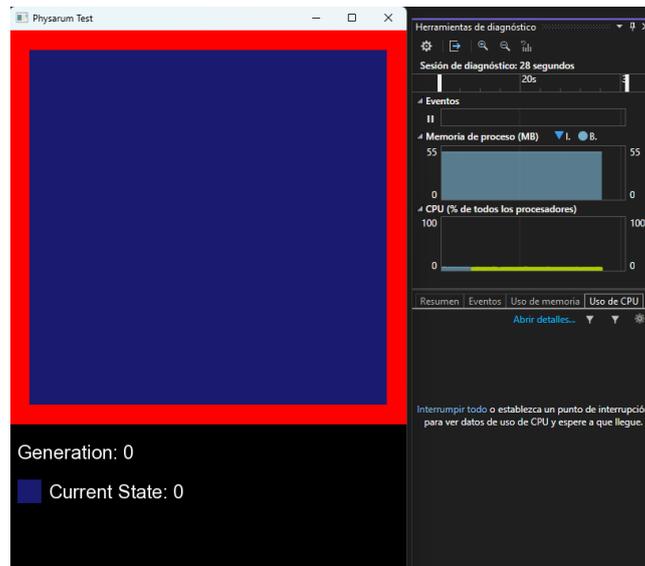


Figura 51: Rendimiento de la aplicación sin estados adicionales en Windows

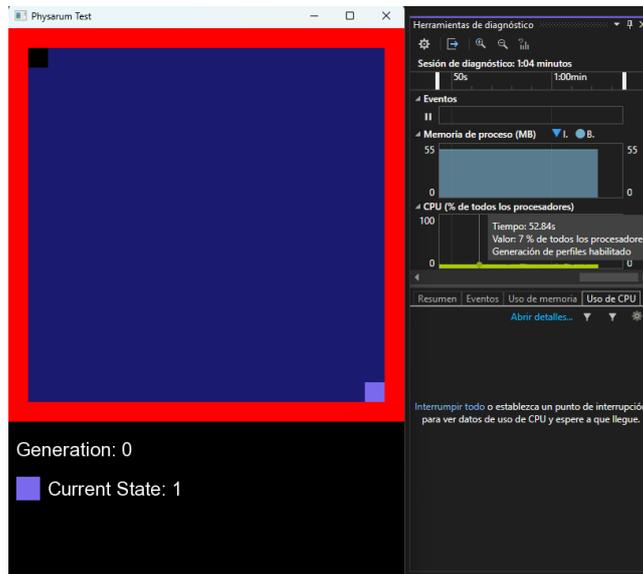


Figura 52: Rendimiento de la aplicación con una configuración inicial en Windows

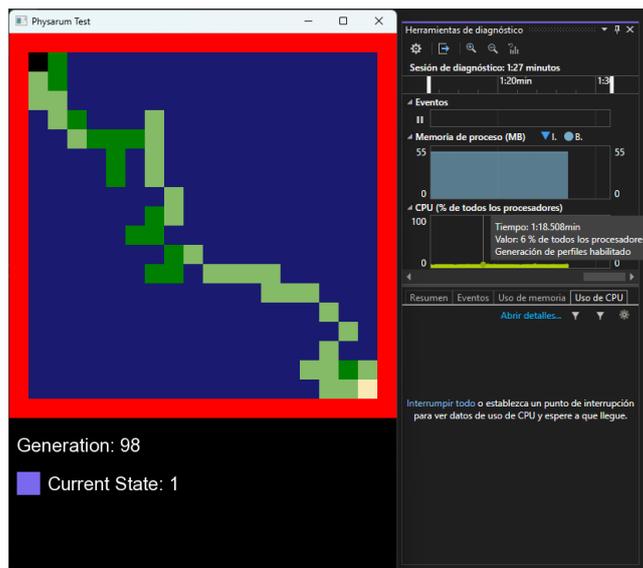


Figura 53: Rendimiento al obtener la ruta en Windows

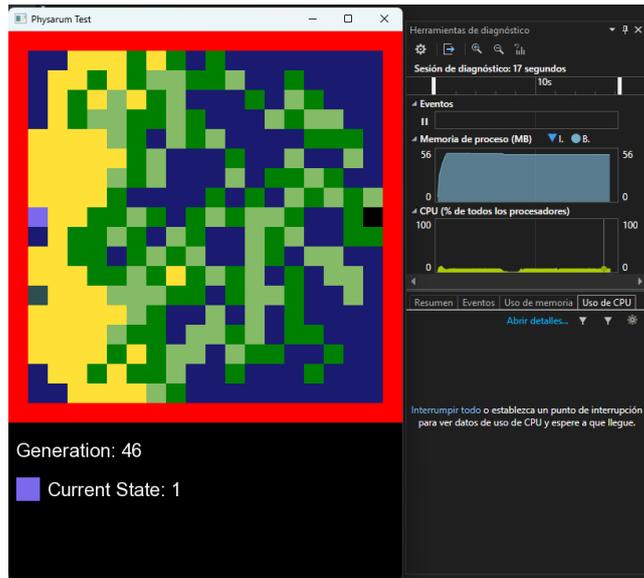


Figura 54: Rendimiento con configuración distinta a la anterior mientras se expande en Windows

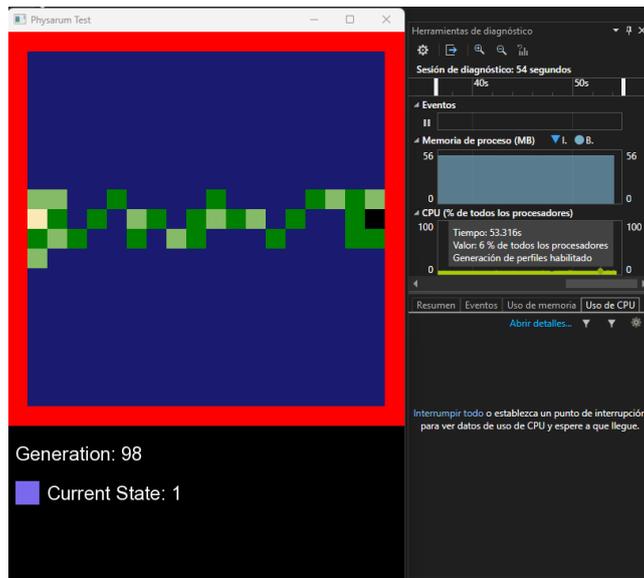


Figura 55: Rendimiento al obtener una ruta en otra configuración

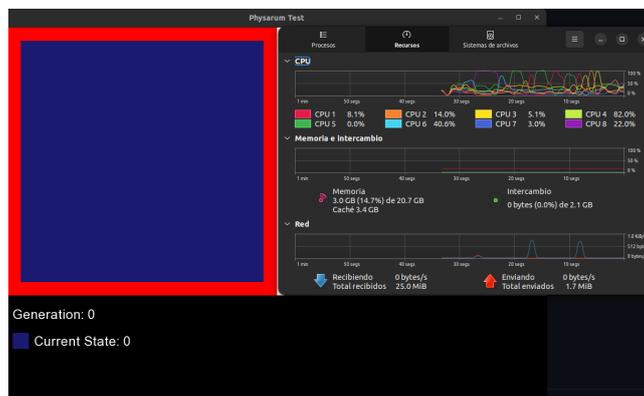


Figura 56: Rendimiento de la aplicación al iniciar el programa en Linux

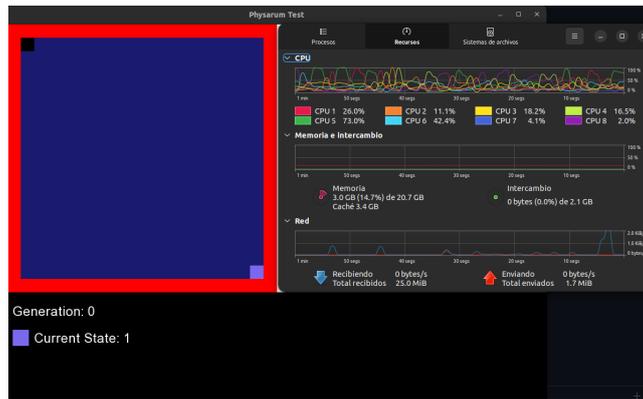


Figura 57: Rendimiento de la aplicación con una configuración inicial en Linux

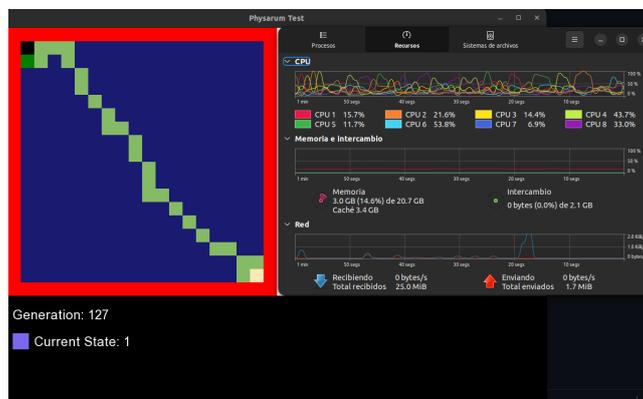


Figura 58: Rendimiento al obtener la ruta en Linux



Figura 59: Rendimiento de la aplicación con un estado inicial diferente en Linux

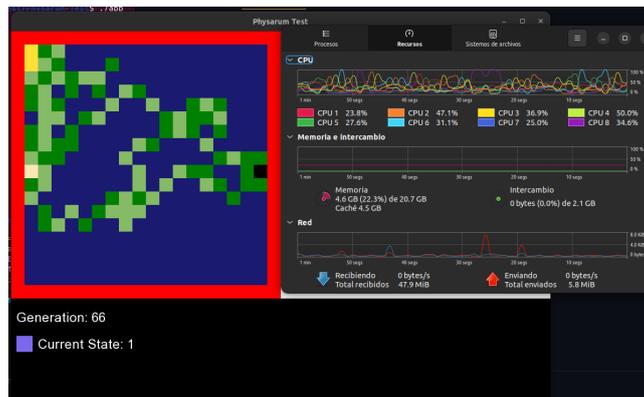


Figura 60: Rendimiento de la aplicación durante expansión de Physarum en Linux

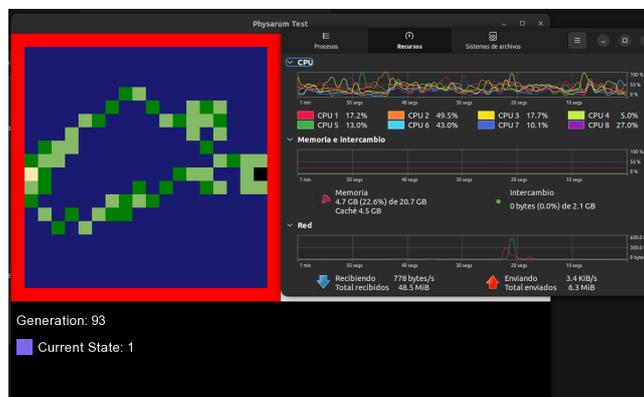


Figura 61: Rendimiento al obtener una ruta en otra configuración en Linux

6.4. Redacción del informe técnico inicial basado en la Iteración 1

Con la información que hasta el momento se ha recabado, se comienza con la redacción del reporte técnico, en la cual quedan asentados varios aspectos que fueron siendo encontrados durante las pruebas de funcionamiento, así como los resultados arrojados por las pruebas unitarias aplicadas al programa.

Además, se tiene en consideración la recopilación de información obtenida a partir de la evaluación del desempeño con algunas de las pruebas iniciales de funcionamiento aplicadas en el programa.

A demás en esta iteración se pudo comprobar el funcionamiento en general del Physarum, obteniendo así el funcionamiento principal de este, el cual es obtener una ruta a partir de un punto inicial y de un punto final. Es importante ver que debido a que es la primera iteración, han surgido muchas observaciones las cuales son de vital importancia para el análisis y mejora del programa, por lo que eso significa que mientras más se avance, el programa estará en un proceso de mejora continua.

Principalmente lo que se pudo notar es el funcionamiento en el caso base, que es el de generar la ruta a través de distintas configuraciones del estado inicial, siendo obtenidas rutas las cuales, son las que el robot interpretaría para poder realizar su funcionalidad de moverse de un lugar a otro.

En los distintos escenarios y pruebas que fueron realizadas en este proceso, se obtuvieron distintos resultados los cuales nos pueden dar una idea sobre el rendimiento y ventanas de oportunidad para la mejora del algoritmo, esto es debido a que se plantearon escenarios y pruebas cuyos resultados son reveladores y que ayudarían a obtener mejores resultados para el objetivo principal del programa, el cual es generar rutas.

6.5. Pruebas unitarias del algoritmo en simulación de mapas 2

Las pruebas realizadas, como ya se había mencionado anteriormente, son realizadas por medio del framework de Google Test hecho para programas que estén escritos en el lenguaje de programación C++.

En estas pruebas, se realizaron nuevas pruebas unitarias a algunas funciones que cambiaron un poco debido al desarrollo y los resultados de algunas pruebas que fueron realizadas en etapas anteriores. Se muestran algunas de las+ pruebas unitarias realizadas en la Figura 62.

Las funciones mostradas son las que fueron agregadas al archivo de pruebas unitarias que ya se habían realizado junto con las nuevas aplicadas debido a la introducción de nuevas funcionalidades dentro del programa. Como podemos ver, todas las pruebas resultaron exitosas, lo que es esperable al menos en su mayoría, ya que como se mencionó anteriormente en las primeras pruebas unitarias, el programa funciona a partir de un modelo matemático y debido a su naturaleza, el funcionamiento depende de que la implementación de ese modelo matemático a través del algoritmo sea correcta.

Las nuevas funciones corresponden a la implementación de la lectura de imágenes a transformar en mapas al inicio del programa, lo que facilita la entrada de entornos en los cuales se quiera simular al Physarum.

```

24 // Demonstrate some basic assertions.
25 TEST(phy_functions, SomeFunctions) {
26     EXPECT_EQ(0, phy_app.allNutrientsFounded()); //Test OK
27     EXPECT_EQ(0, phy_app.setCellState()); //Test OK
28     EXPECT_EQ(0, phy_app.cleanCells(sf::Time)); // Test OK
29     EXPECT_EQ(0, phy_app.showPhysarum()); // Test OK
30     EXPECT_EQ(0, phy_app.evaluatePhysarum()); // Test OK
31     EXPECT_EQ(0, phy_app.physarumTransitionConditions()); // Test OK
32     EXPECT_EQ(0, phy_app.getRoute()); // test ok
33
34 // Demonstrate some basic assertions.
35 TEST(Loadmap, SomeFunctions) {
36     EXPECT_EQ(0, loadmap.convertImageToMap()); //Test OK
37     EXPECT_EQ(0, loadmap.setDataToArray()); //Test OK
38     EXPECT_EQ(0, loadmap.grayscaleImage(sf::Time)); //Test OK
39     EXPECT_EQ(0, loadmap.setDataToArray()); //Test OK
40
41
42 // Demonstrate some basic assertions.
43 TEST(autotest, SomeFunctions) {
44     EXPECT_EQ(0, loadmap.getRandom(const int, const int)); //Test OK
45
46 }
47

```

Figura 62: Prueba unitaria 2

6.6. Pruebas de aceptación en escenarios pequeños

6.6.1. 1ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado.

Flujo: El programa es cargado.

Se le presenta el lienzo en pantalla.

Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección.

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.6.2. 2da Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.6.3. 3ra Prueba de aceptación: Inicialización de la simulación.

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.7. Evaluación de desempeño del software en pruebas iniciales 2

En las pruebas iniciales ya en entornos controlados y de un tamaño bastante reducido, se hizo del conocimiento de algunos detalles que surgen de la implementación de algunas funcionalidades, las cuales fueron revisadas a detalle.

Como se mencionó anteriormente, como medida de desempeño se utilizará el tiempo que tarda el software entre el cambio de generación entre una evaluación de las reglas y otra. A diferencia de las anteriores evaluaciones, en este caso al ya implementar un mapa que está basado ya en una locación en específico, se tiene que realizar una configuración la cual sea lo más aproximado a su locación en el mundo real.

Las siguientes pruebas corresponden a un mapa el cual se tiene por defecto y representa en el área en la cual el robot fue colocado en la vida real para poder comprobar el funcionamiento del seguimiento de la ruta. Es importante recalcar que las pruebas se realizan en ambos sistemas operativos, los cuales son, como es asumido desde el inicio, el sistema operativo de Windows y Linux, usando como principal distribución la de Ubuntu en el caso de Linux. Esto se se puede ver en las Figuras 63 - 68.

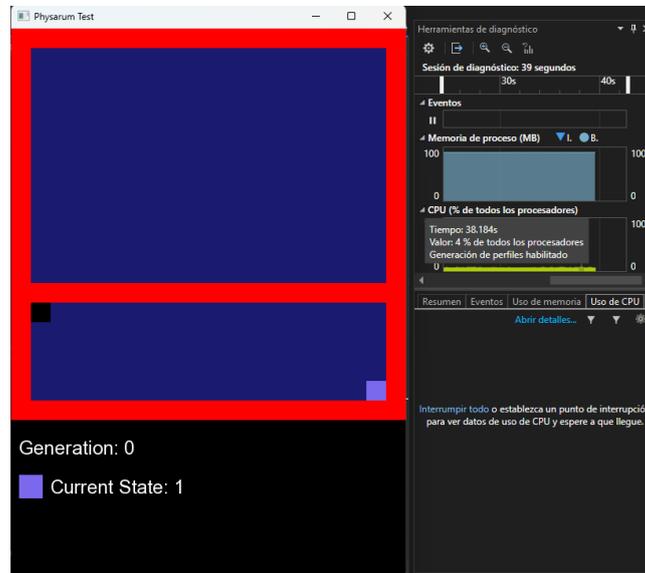


Figura 63: Estado inicial con una barrera simulando un cuadro cerrado en Windows

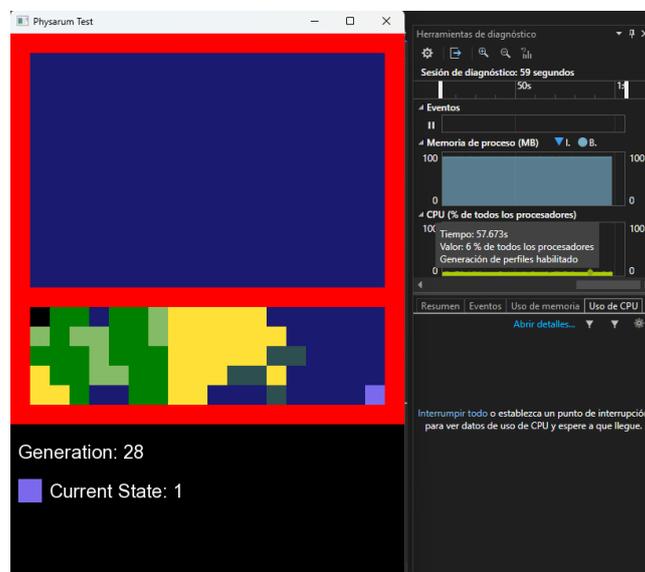


Figura 64: Rendimiento de la aplicación en un entorno cerrado y durante la expansión del Physarum en Windows

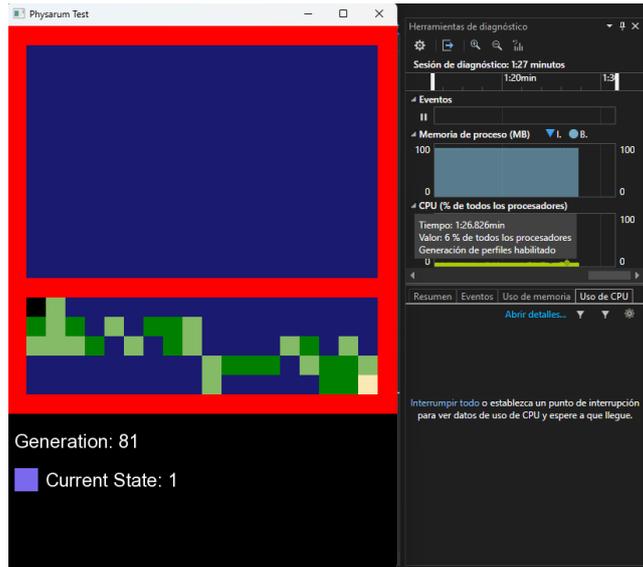


Figura 65: Rendimiento al generar la ruta con el Physarum Windows



Figura 66: Rendimiento de la aplicación durante el establecimiento del estado inicial en Linux

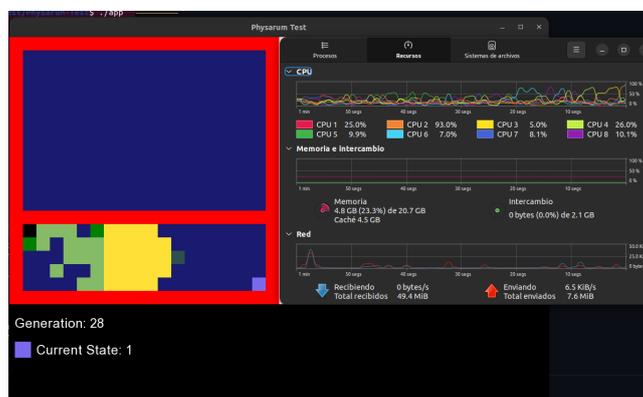


Figura 67: Rendimiento de la aplicación durante la expansión del Physarum en Linux

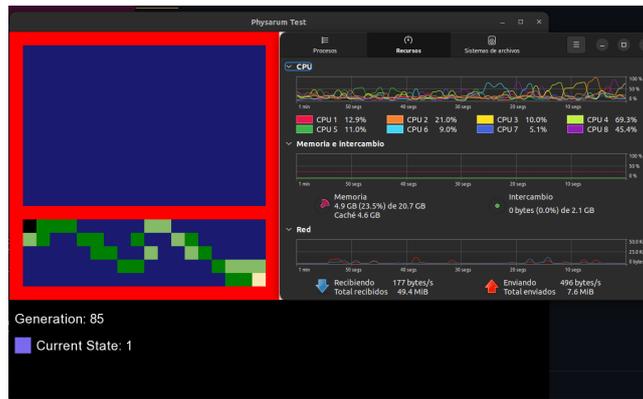


Figura 68: Rendimiento al generar la ruta con el Physarum en Linux

6.8. Recopilación de datos del software y desempeño 2

Los datos que han sido recopilados hasta el momento con los que están en base a las pruebas con los mapas que representan ya un escenario de la vida real. Como se puede apreciar en la Figura 69 hay distintos estados que se pueden colocar en el lienzo del estado inicial antes de disparar el simulador para obtener la ruta. Uno de estos estados es el estado 2, el cual representa un repelente que, a efectos prácticos de nuestro software, son la representación de lo que fueran paredes u obstáculos en el entorno o el mapa que se coloca como estado inicial.

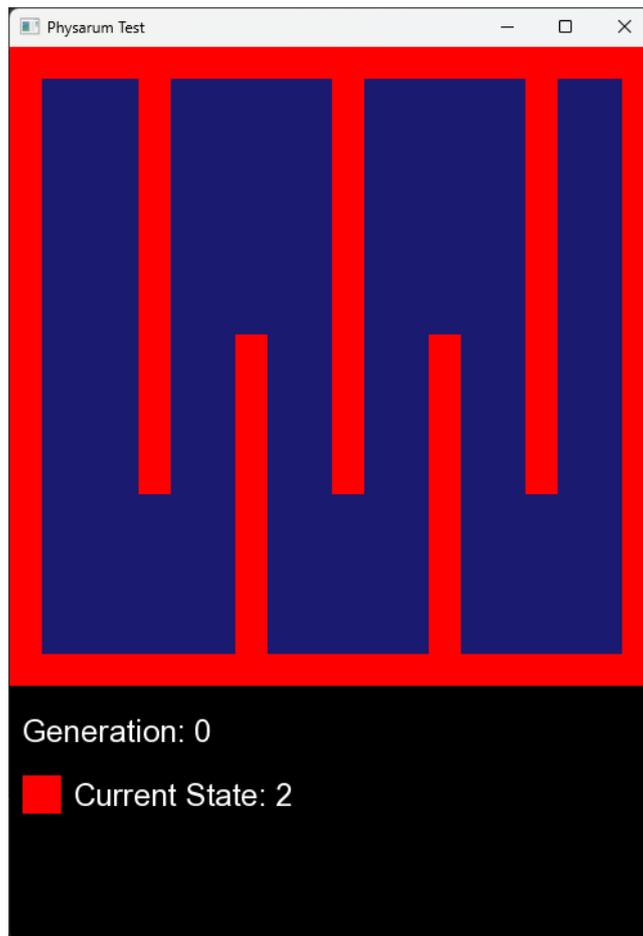


Figura 69: Una configuración inicial con obstáculos

El repelente tiene como función en el simulador, impedir el paso del Physarum en los puntos donde este se encuentre, por lo que funciona bien como si fuera la representación de paredes o barreras de los mapas en donde no se puede avanzar en la vida real.

Así que esta vez, las pruebas para medir el desempeño, que como anteriormente se mencionó, se hacen midiendo el tiempo entre el avance de un estado y el siguiente, fueron sobre un estado inicial donde se coloca inicialmente una serie de obstáculos para llegar al destino final, esto es como si fuera un laberinto por el cual el Physarum debe de pasar para poder llegar a su destino. Esto hace que la evaluación en general tarde un poco más que si fuera un mapa o estado inicial sin algún tipo de obstáculo.

De las Figuras 70 - 77, se muestran algunos de los resultados obtenidos con algunas configuraciones.

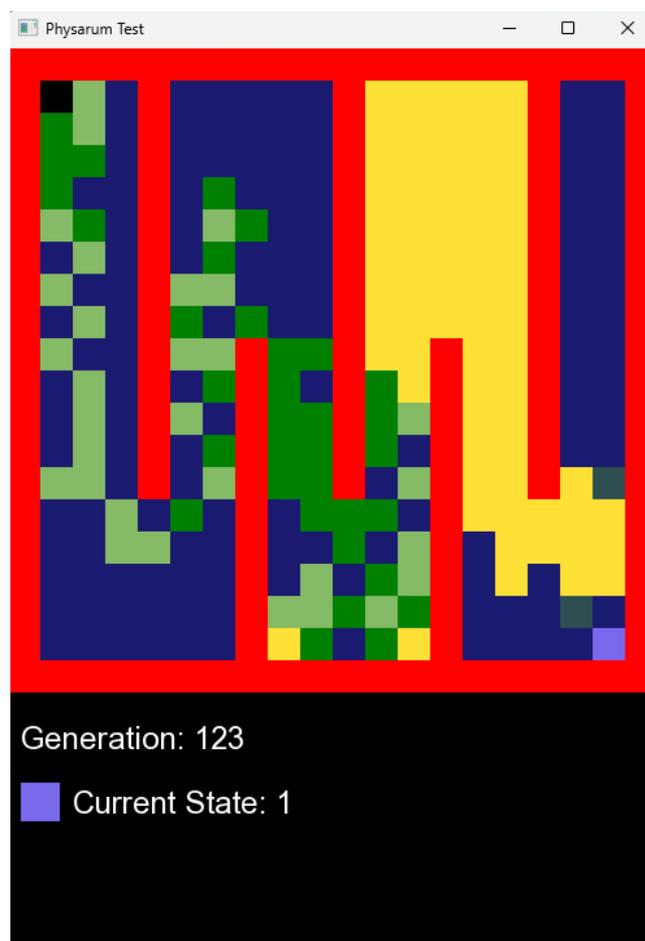


Figura 70: Expansión del algoritmo de Physarum.



Figura 71: Ruta generada a partir de una configuración inicial complicada.

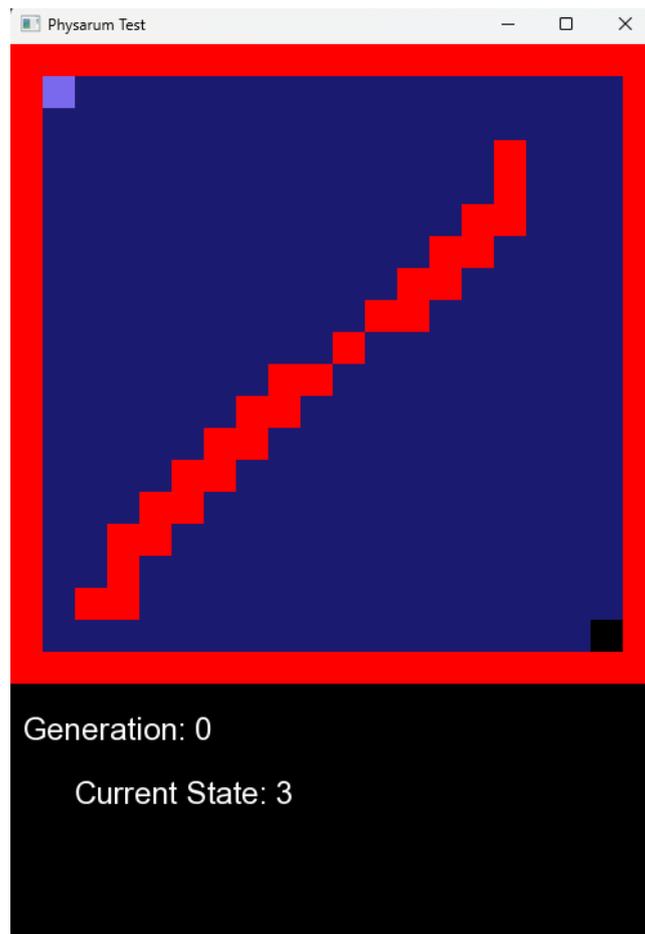


Figura 72: Estado inicial de una configuración.

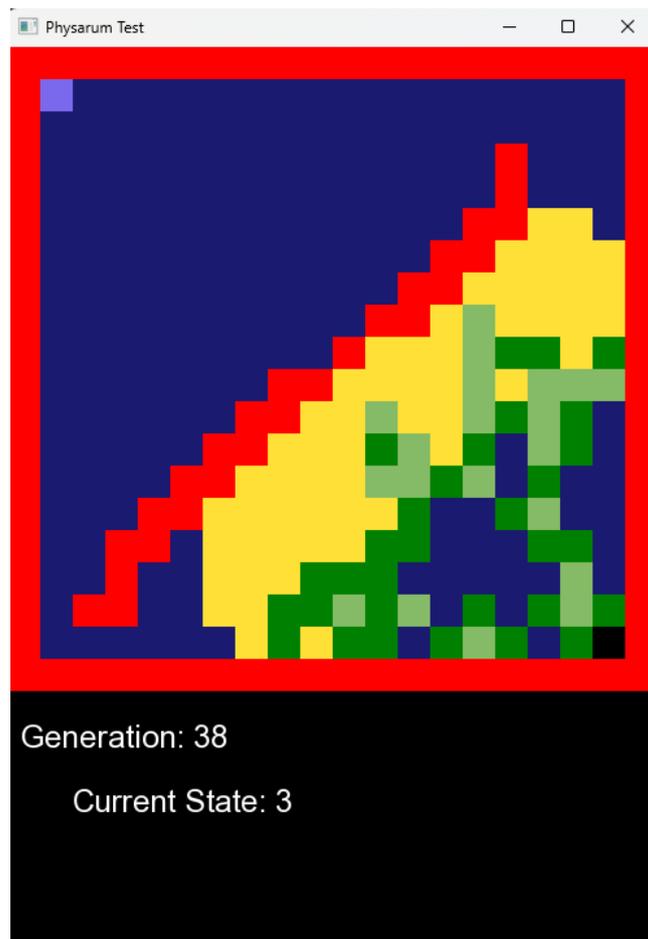


Figura 73: Expansión del algoritmo del Physarum con una pared diagonal en medio.

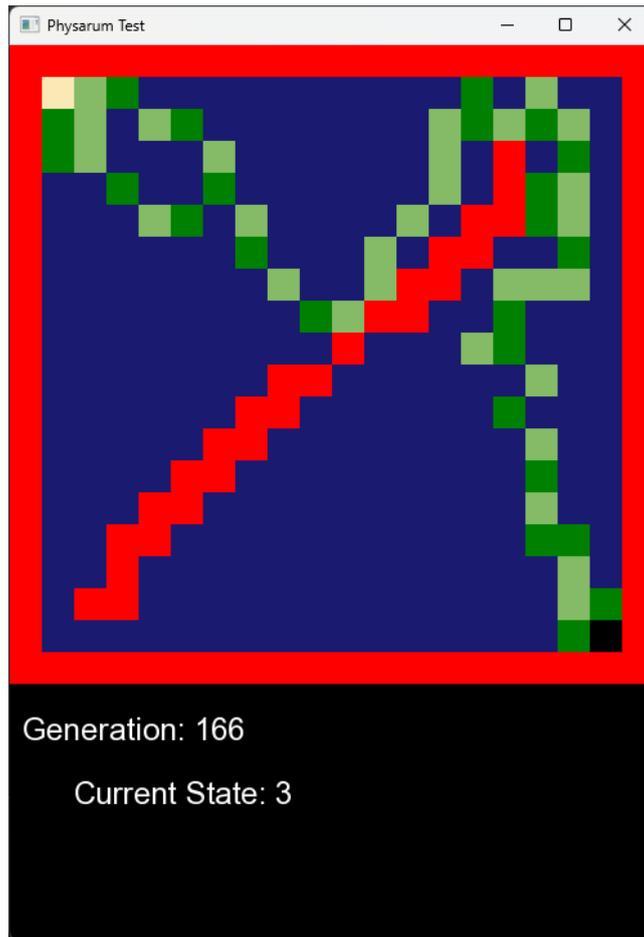


Figura 74: Generación de la ruta satisfactoria que es generada durante la ejecución y finalización del algoritmo.

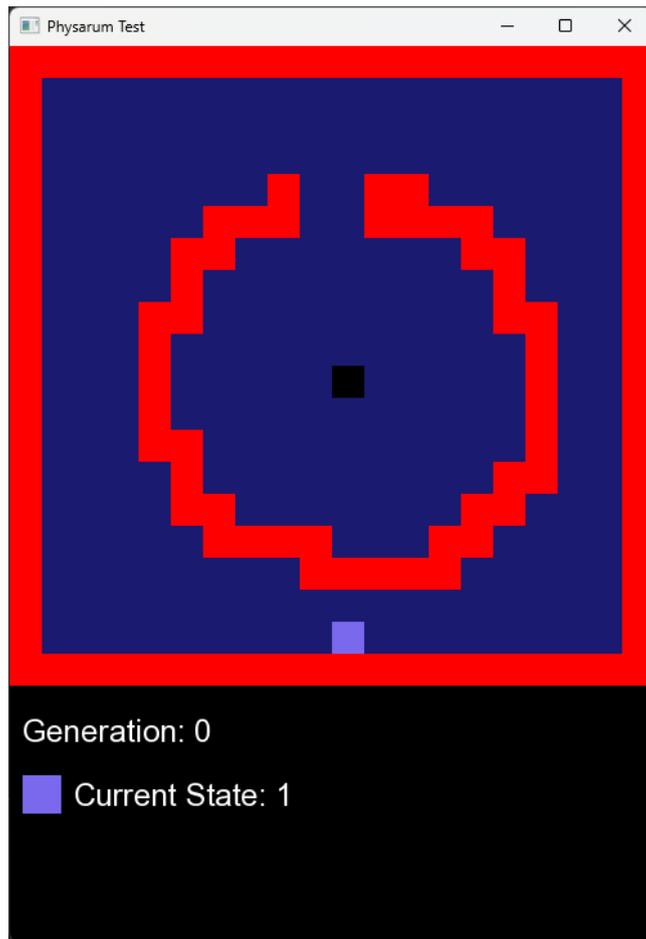


Figura 75: Estado inicial con configuración circular.

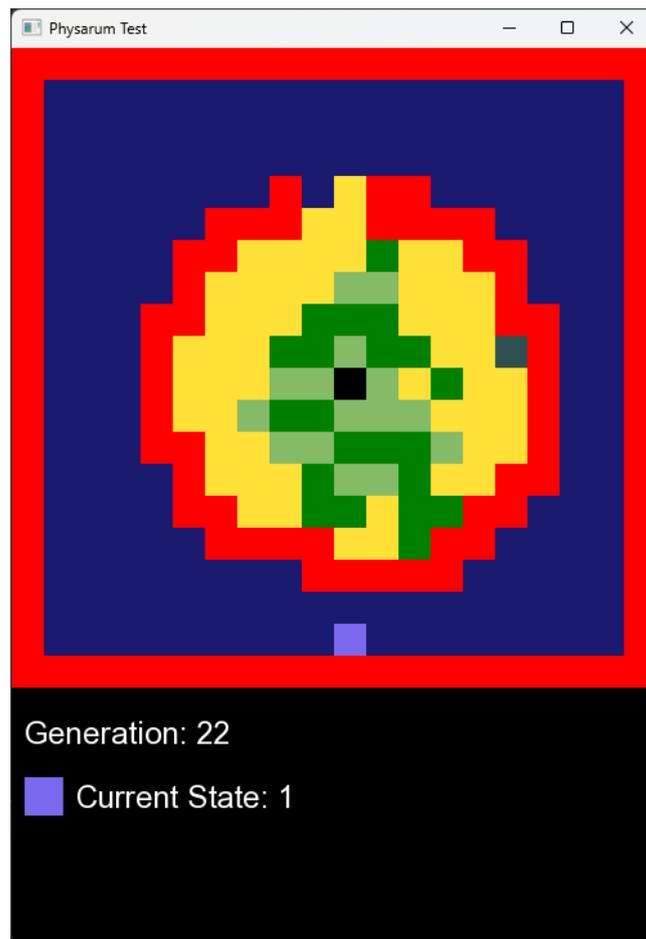


Figura 76: Expansión con una configuración de repelentes circular.

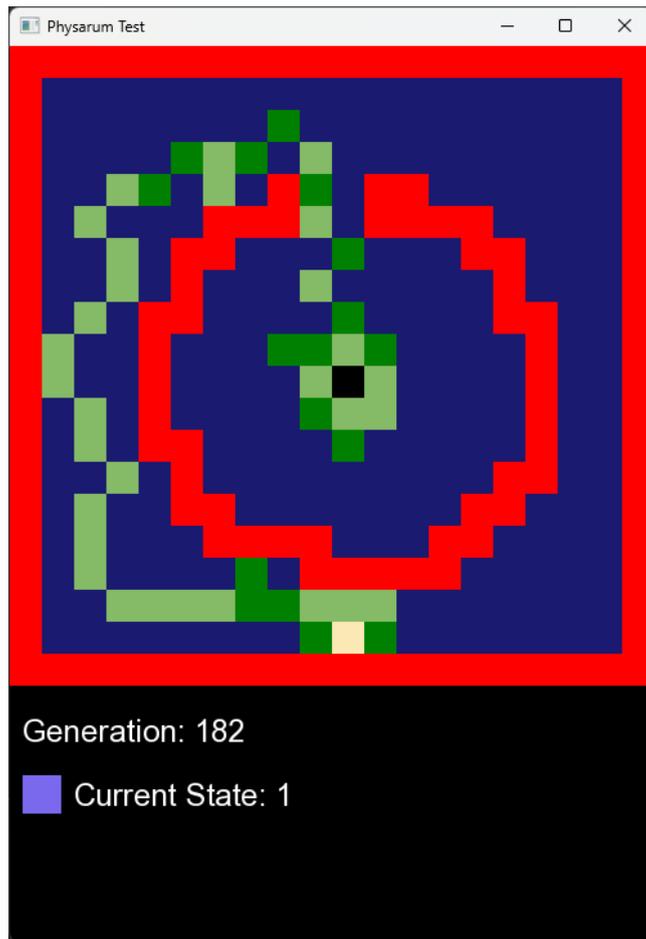


Figura 77: Ruta generada después de la finalización de la simulación.

6.9. Pruebas de aceptación en entornos medianos

Para la realización de estas pruebas, se utilizaron exactamente los mismos parámetros de las pruebas unitarias anteriores, ya que la única diferencia es que el lienzo es mucho más grande, por lo que esta prueba es agregada también.

6.9.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión $n \times n$ correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.9.2. 2da Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado.

Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección.

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.9.3. 3ra Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.9.4. 4ta Prueba de aceptación: Inicialización de la simulación.

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.10. Pruebas de aceptación en entornos complejos simulados 1

Estas pruebas comprenden a las anteriores pruebas, ya que al igual que las pruebas de aceptación de la iteración anterior, para el entorno más complejo, el tamaño de lienzo es lo único que cambia, pero lo único que se agrega ahora es la carga de un mapa de un entorno real.

6.10.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión $n \times n$ correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código.
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.10.2. 2da Prueba de aceptación: Carga de mapas

Descripción: En el código se cargan imágenes, las cuales corresponden a mapas, donde éstas se cargan para colocarlos en el lienzo al iniciar el programa

Flujo:

- Se coloca la dirección de la imagen
- Se coloca un tamaño para tanto la imagen como para el lienzo
- Se inicia el programa.
- El programa al iniciar, carga la configuración, mostrando la imagen del mapa, convertido al lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.

- Se pueden colocar estados en cualquier área correspondiente al lienzo.
- La imagen del mapa es colocada correctamente en el lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.10.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado. Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.10.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.10.5. 5ta Prueba de aceptación: Inicio de la simulación

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.11. Evaluación final del desempeño del software 1

Para la evaluación del desempeño, en esta ocasión fue mucho más claro la medida del tiempo transcurrido entre el paso de una generación y otra, y esto es debido al tamaño en el cual crece el arreglo de dimensión $n * n$, ya que mientras se tenga un n mucho más alto, el rango de evaluación también crece considerablemente y también lo hacen las evaluaciones en cada una de las células, donde las reglas del algoritmo del Physarum deben de hacer posible la transición entre cada uno de los estados.

De las Figuras 78 - 81, se muestran algunas de las configuraciones junto con sus resultados y los datos recopilados en tamaños mucho más extensos:

La evaluación final del desempeño del software se centró en medir el tiempo de simulación y la precisión de las rutas generadas bajo diversas condiciones iniciales y sistemas operativos. Se comprobó que el algoritmo, aunque eficiente en la mayoría de los casos, presenta diferencias de desempeño notables entre Windows y Linux, con mejores resultados en Linux debido a su mayor capacidad de procesamiento en simulaciones intensivas.

El análisis de los tiempos de ejecución entre una generación y la siguiente permitió identificar mejoras clave en la eficiencia del software. Los resultados mostraron que, en entornos de mayor complejidad, el tiempo de procesamiento incrementa proporcionalmente al número de barreras y obstáculos, lo que requiere optimizaciones adicionales en futuras iteraciones.

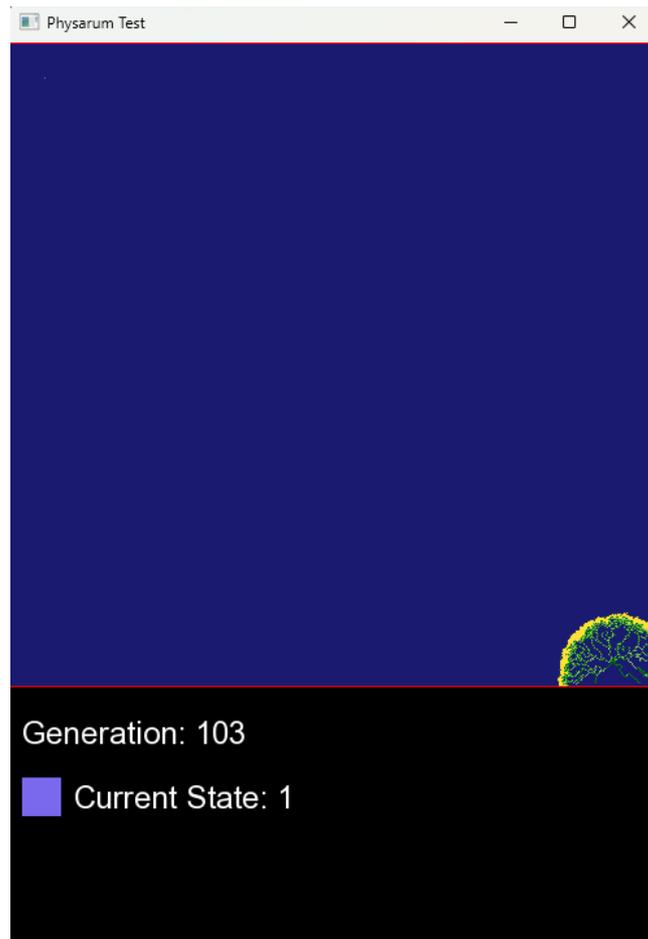


Figura 78: Expansión y evaluación de la simulación en un entorno mucho mas grande (500 x 500)

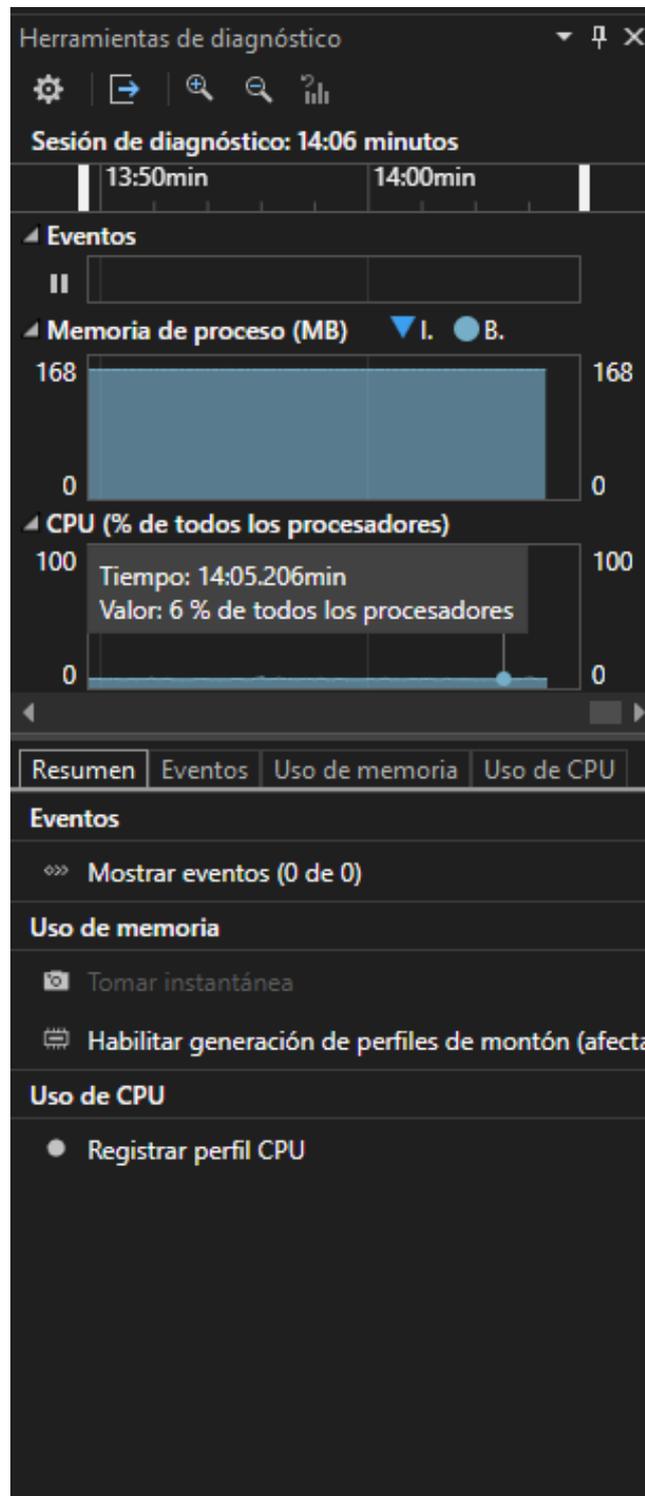


Figura 79: Cantidad de memoria y procesamiento durante la ejecución del simulador



Figura 80: Expansión en el mismo tamaño del que fue realizado en Linux.

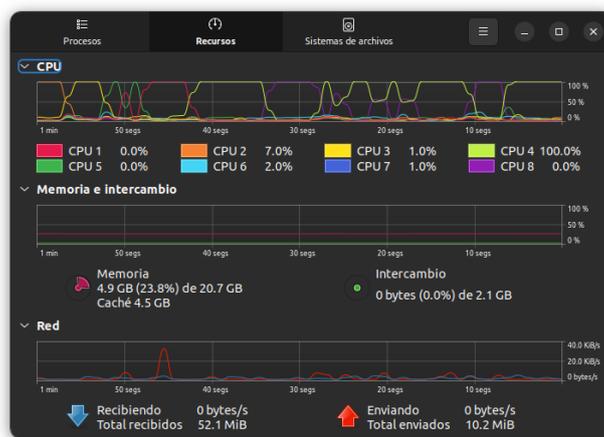


Figura 81: Uso de los recursos del computador durante la ejecución del simulador.

6.12. Recopilación y análisis de datos del rendimiento del robot de IEEE 1872.1 e IEEE 2914

En el mundo de las pruebas y otro tipo de documentación, son importantes las normas, debido a que permiten un estándar para el manejo de distintos elementos dentro de un sistema, además de que la estandarización fomenta el intercambio sencillo o interacción entre componentes los cuales no tienen mucho que ver con otros, pero que al existir una norma, un modo de estandarización, se pueden crear formas en las que ambas partes puedan coexistir entre sí y entre muchas otras más.

La importancia de la IEEE 1872.1 radica en que esta norma nos permite definir ciertos elementos y contextos en los cuales se puede realizar un intercambio de información entre pares, los cuales usualmente están relacionados a la electrónica y robótica, así como al hardware y software. En el caso del software y del robot al cual el software le genera una ruta para su posterior seguimiento, usa criterios de interoperabilidad que son los más comunes a la hora de poder conectar los resultados obtenidos por el programa, así como la forma de interpretar esos datos para cumplir el objetivo principal del algoritmo de Physarum: generar una ruta la cual el robot pueda seguir sin problemas para llegar a un destino final.

Para el caso de el envío de información entre el robot y el software, se hace uso de herramientas FTP, así como de un servidor el cual recibe peticiones HTTP con las cuales se pueden recibir y enviar los datos que sean necesarios.

A pesar de que el software pertenezca a una de las jerarquías más bajas dentro de la estructura de la IEEE, es de vital importancia que éste último funcione correctamente y cumpla su función para que el robot tenga un propósito y pueda tener la utilidad necesaria.

Y dentro de las demás normas, también se toma en cuenta la norma IEEE 2914, que dentro de la parte generadora de la ruta y demás trabajo del software de ruteo a partir del Physarum, se vuelve importante tomarla en cuenta. Esto porque a diferencia de otros sistemas donde la toma de decisiones trata de ser siempre la óptima y en muchos casos están se pueden llegar a replicar para tener un registro y una idea de que decisiones te llevan a un determinado resultado, en el caso del simulador del Physarum las decisiones son aleatorias en todo momento, además, de que la aleatoriedad es aplicada a todos los elementos que componen al arreglo de dos dimensiones, por lo que se vuelve muy difícil de rastrear y obtener siempre algún mismo resultado, puesto que en cada ejecución del algoritmo, la configuración y resultados obtenidos serán distintos, por lo que lo único que se debe de revisar es que el resultado sea en su mayoría satisfactorio, que a pesar de que no siempre la ruta sea la más óptima, se acerque a hacerlo, para así generar una mejor movilidad y toma de decisiones y tratamiento de la información.

La interoperabilidad entre sistemas robóticos se beneficia del uso de normas como la IEEE 1872.1, que no sólo define términos y conceptos, sino también los contextos necesarios para asegurar un intercambio efectivo de datos. Esto implica establecer conexiones claras entre los sensores, el software de control, y el hardware robótico, lo que facilita una sincronización continua y un flujo de datos eficiente.

En el marco de la evaluación de datos del rendimiento, es crucial contar con mecanismos que validen la integridad y calidad de la información obtenida por el robot. Se hace especial énfasis en la consistencia de los datos recogidos, el manejo de errores y la representación fidedigna del comportamiento del robot en escenarios controlados y reales, donde las normas IEEE ayudan

a uniformar estos criterios.

Un aspecto relevante de la aplicación de la norma IEEE 2914 radica en la gestión del comportamiento estocástico del algoritmo Physarum. Debido a la variabilidad en las rutas generadas, es necesario implementar mecanismos de observación y registro que permitan un análisis comparativo entre ejecuciones, estableciendo patrones y tendencias que puedan mejorarse en iteraciones futuras.

La generación de rutas mediante el algoritmo de Physarum implica una interacción constante entre elementos aleatorios y deterministas, los cuales deben evaluarse bajo criterios de eficacia y seguridad. Las normas IEEE proporcionan las pautas para medir y asegurar que, aunque los resultados puedan variar, las soluciones propuestas mantengan un grado aceptable de estabilidad y funcionalidad.

La integración de herramientas de comunicación, como servidores FTP y protocolos HTTP, dentro del entorno robótico asegura un manejo eficiente de los datos. Este proceso debe regirse por estándares que garanticen la seguridad, la velocidad y la precisión del intercambio de información, tanto en ambientes controlados como en situaciones en las que el robot interactúe con diferentes dispositivos.

Dado que el software que rige el comportamiento del robot actúa como un intermediario crítico entre los sensores y el hardware, su correcto funcionamiento es fundamental. La norma IEEE 1872.1 se encarga de establecer un conjunto de reglas para garantizar que esta comunicación sea efectiva, asegurando que todos los componentes involucrados operen de manera conjunta y coherente.

El éxito en la generación de rutas óptimas o satisfactorias mediante el modelo de Physarum depende, en gran medida, de la capacidad del sistema para adaptarse a condiciones cambiantes. Este enfoque flexible se encuentra respaldado por la norma IEEE 2914, que permite explorar diversas configuraciones y evaluar su eficacia en tiempo real, promoviendo mejoras continuas y soluciones creativas dentro del contexto robótico.

6.13. Redacción del informe con mejoras de la Iteración 2

En esta etapa se realizó un análisis exhaustivo de los resultados obtenidos en la Iteración 2, enfocándonos en la identificación de áreas clave de mejora en el software. A partir de las pruebas realizadas y los datos recopilados, se implementaron ajustes en el algoritmo para optimizar su rendimiento en entornos más complejos. Estos ajustes se centraron en mejorar la eficiencia en la generación de rutas, reduciendo los tiempos de procesamiento en mapas de mayor tamaño y con más obstáculos.

Además, se revisaron y actualizaron las pruebas unitarias para asegurar la correcta implementación de las mejoras, enfocándonos en la estabilidad del sistema al enfrentarse a diferentes configuraciones iniciales. Los cambios implementados lograron una reducción significativa en los tiempos de simulación, especialmente en los casos donde se incluían múltiples barreras o rutas más largas.

Asimismo, se refinaron las estrategias para eliminar células innecesarias en las rutas generadas, lo que permitió un recorrido más eficiente por parte del robot, optimizando su capacidad de seguir las coordenadas calculadas por el simulador. Estos avances fueron documentados y

evaluados, dejando asentadas las mejoras que serán fundamentales para las siguientes iteraciones.

El informe incluye un análisis detallado de los tiempos de ejecución, las rutas generadas en diversas configuraciones y las métricas de desempeño obtenidas en sistemas operativos tanto Windows como Linux. Estos resultados muestran un progreso claro en el desarrollo del software, destacando las mejoras implementadas para su robustez y precisión en escenarios más exigentes.

6.14. Recopilación final de datos del software para el informe 1

Los datos hasta ahora recopilados nos muestran un panorama mucho más amplio del que se tenía al inicio de la realización y desarrollo del software, esto es, se han recopilado bastantes datos los cuales ayudan a que se pueda encontrar una mucho mejor forma de realizar algunas de las funcionalidades o se hayan encontrado algunas carencias de las cuales muchas veces pasan desapercibidas en una primera instancia.

El programa, como vimos al inicio, funciona correctamente y cumple su función de generar rutas, no obstante, las rutas muchas veces tienden a ser un poco imperfectas debido a la propia naturaleza del algoritmo en sí, además de que muchas veces se generan bifurcaciones o hay demasiadas células en determinadas partes de la ruta, lo que fomenta que se encuentre una mayor dificultad a la hora de generar una ruta óptima para ser enviada al robot, además de que el proceso de generación de coordenadas se complica debido a que cada una de estas debe de tener algún tipo de orden el cual ayude a que el robot tenga una forma mucho más fácil de moverse en su entorno real y que lo haga sin tener que dar muchas vueltas.

Por lo que se empezaron a generar soluciones, como lo es limpiar y eliminar algunas de las células dentro del arreglo con el objetivo de optimizar la ruta, además de generar algoritmos los cuales puedan tener una mejor aproximación al correcto ordenamiento de las células presentes y que al final, la ruta sea una la cual el robot pueda seguir sin problemas y que, respecto a la generada inicialmente por el simulador del Physarum, sea mucho más conveniente. Por último, se pudo notar que mientras más incrementaba el tamaño del lienzo, el programa tiende a alentarse un poco más, esto es debido a la gran cantidad de información que tiene que desplegar al mismo tiempo y dificulta el correcto manejo del software, por lo que en las futuras implementaciones e iteraciones se tendrá que mejorar este aspecto para así poder optimizar los recursos que son usados y tener una mejor experiencia a la hora de usar el simulador y así, tener un mejor confort a la hora de generar una ruta.

6.15. Documentación del avance segunda iteración

Durante la segunda iteración, se realizaron mejoras significativas en la funcionalidad del simulador y su capacidad para generar rutas más óptimas y adaptarse a entornos más complejos. Se implementaron ajustes en el algoritmo basado en el Physarum Polycephalum para reducir los tiempos de simulación y mejorar la precisión en la selección de rutas. Además, se realizaron pruebas exhaustivas en distintos sistemas operativos (Windows y Linux), documentando las diferencias de rendimiento y comportamiento del software en ambos entornos.

En esta iteración, también se introdujeron mejoras en la interfaz de control del robot, facilitando el seguimiento de rutas generadas y la interacción con los sensores del robot, como el sistema LiDAR. La documentación incluye los cambios implementados, los resultados obtenidos y las estrategias propuestas para las pruebas de aceptación en entornos complejos, que servirán

de base para la próxima iteración.

Durante la segunda iteración, se realizaron mejoras significativas en la funcionalidad del simulador y su capacidad para generar rutas más óptimas y adaptarse a entornos más complejos. Se implementaron ajustes en el algoritmo basado en el Physarum Polycephalum para reducir los tiempos de simulación y mejorar la precisión en la selección de rutas. Además, se realizaron pruebas exhaustivas en distintos sistemas operativos (Windows y Linux), documentando las diferencias de rendimiento y comportamiento del software en ambos entornos.

En esta iteración, también se introdujeron mejoras en la interfaz de control del robot, facilitando el seguimiento de rutas generadas y la interacción con los sensores del robot, como el sistema LiDAR. La documentación incluye los cambios implementados, los resultados obtenidos y las estrategias propuestas para las pruebas de aceptación en entornos complejos, que servirán de base para la próxima iteración.

6.16. Pruebas de aceptación en entornos complejos simulados 2

Para validar el rendimiento del software en escenarios más desafiantes, se llevaron a cabo pruebas de aceptación en entornos complejos simulados. En estas pruebas, se introdujeron obstáculos adicionales y laberintos más elaborados en el lienzo de simulación, lo que permitió observar el comportamiento del algoritmo ante situaciones de alta densidad de barreras.

Los resultados mostraron que el algoritmo es capaz de adaptarse a estos entornos, generando rutas eficientes incluso con una mayor presencia de obstáculos. Sin embargo, se identificaron áreas donde el tiempo de cálculo aumenta significativamente en mapas de gran tamaño, lo que sugiere la necesidad de optimizar ciertos componentes del algoritmo en futuras iteraciones.

Estas pruebas fueron realizadas en entornos simulados de tamaño más grande, y los resultados revelaron que las rutas generadas continúan siendo precisas, aunque con un mayor tiempo de procesamiento, especialmente en sistemas con recursos limitados.

6.17. Evaluación final del desempeño del software 2

La evaluación final del desempeño del software se centró en medir el tiempo de simulación y la precisión de las rutas generadas bajo diversas condiciones iniciales y sistemas operativos. Se comprobó que el algoritmo, aunque eficiente en la mayoría de los casos, presenta diferencias de desempeño notables entre Windows y Linux, con mejores resultados en Linux debido a su mayor capacidad de procesamiento en simulaciones intensivas. Esto se reflejó en las Figuras 82 y 83.

El análisis de los tiempos de ejecución entre una generación y la siguiente permitió identificar mejoras clave en la eficiencia del software. Los resultados mostraron que, en entornos de mayor complejidad, el tiempo de procesamiento incrementa proporcionalmente al número de barreras y obstáculos, lo que requiere optimizaciones adicionales en futuras iteraciones.

6.18. Recopilación final de datos del software para el informe 2

Los datos finales recopilados del software incluyen mediciones de tiempos de ejecución, precisión de rutas generadas y análisis de desempeño en distintos escenarios.

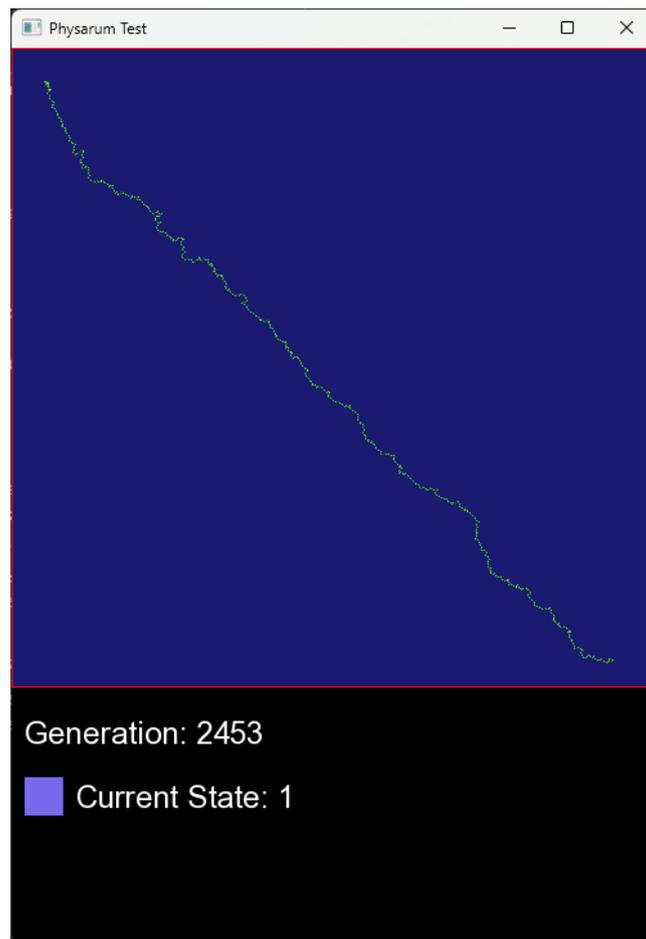


Figura 82: Generación de una ruta después de el paso de generaciones en Windows (500 x 500)

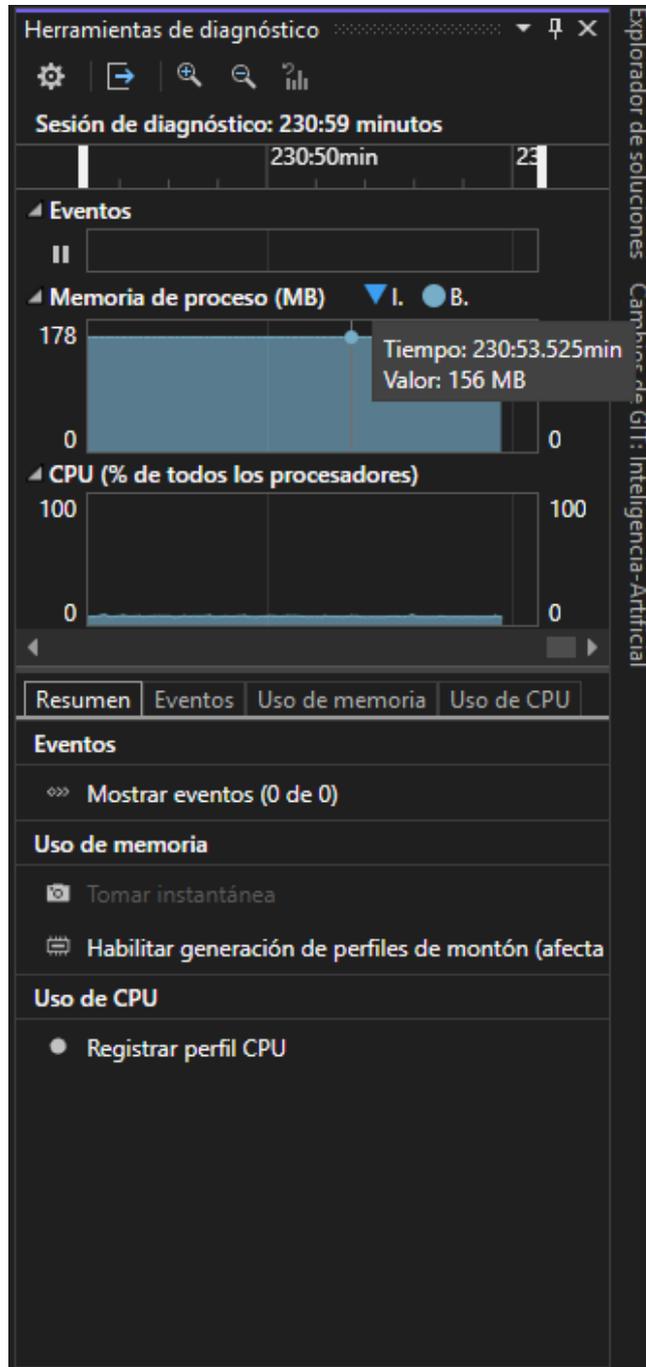


Figura 83: Uso de la memoria y CPU durante la obtención de la ruta del Physarum.

Se realizaron simulaciones en mapas tanto simples como complejos, con variaciones en los obstáculos y configuraciones iniciales, para obtener un conjunto completo de datos que permitieran evaluar el rendimiento general del software.

Además, se documentaron las diferencias entre los resultados obtenidos en Windows y Linux, mostrando cómo el sistema operativo afecta la eficiencia del algoritmo. Estos datos serán fundamentales para justificar las mejoras implementadas y establecer un punto de referencia para las siguientes iteraciones del desarrollo.

6.19. Informe final con resultados de la Iteración 3

El informe final de la Iteración 3 presenta una evaluación detallada del rendimiento del software, con base en los datos obtenidos de las pruebas y simulaciones realizadas. Se documentaron las rutas generadas en escenarios complejos, los tiempos de simulación y las mejoras implementadas para optimizar el desempeño en entornos más exigentes.

Entre los principales resultados, se destaca la capacidad del algoritmo para adaptarse a mapas de mayor tamaño y con múltiples obstáculos, aunque con un incremento en el tiempo de procesamiento. También se destaca la mejora en la limpieza de las rutas generadas, lo que permitió que el robot siguiera las coordenadas con mayor precisión.

Este informe concluye con recomendaciones para la siguiente iteración, enfocadas en optimizar el tiempo de ejecución del software en escenarios más grandes y complejos, así como en continuar mejorando la integración con los sistemas de control del robot para asegurar un seguimiento más preciso de las rutas generadas.

Algunos de los datos recopilados tienen relación con las pruebas realizadas directamente sobre casos de la realidad, donde algunos de los datos que recopilamos, como el comportamiento del robot en un entorno con mucha gente, así como dejando que llegue a una distancia mucho más larga de lo habitual, nos representaron datos de suma importancia para su análisis y reproducción de posibles nuevas pruebas futuras para el software y sus subsecuentes aplicaciones.

Los resultados obtenidos también comprenden a los casos aplicados en la vida real, los cuales representan un gran paso para el uso y posible aplicación del robot en entornos los cuales representen un reto para otro tipo de vehículos o incluso, otro tipo de robots los cuales encuentren cierta dificultad a la hora de realizar las mismas tareas que nuestro robot.

Surgen distintas y varias aplicaciones en las cuales el robot es capaz de actuar en distintos tipos de escenarios, cuyos retos a la hora de navegación sean variados y donde la posibilidad de obtener rutas que sean complicadas, y que con el uso de el simulador del Physarum, podemos solventar varios de estos retos, como lo son la generación de rutas de forma aleatoria, pero que siempre se acercan a una ruta óptima, la constante aparición de obstáculos y además la generación de una nueva ruta a partir de que se sea encontrado un obstáculo el cual originalmente no se encontraba la primera vez que fue generada la ruta.

Por último, los resultados nos funcionan como una bitácora para poder analizar el comportamiento y las características que fueron cambiando durante el desarrollo e implementación del robot, así como una forma de ver qué cambios tuvieron una gran relevancia a la hora de implementarlos y sobre qué otro tipo de cambios son posibles y qué se puede hacer para poder

seguir mejorando tanto el algoritmo, así como la implementación en el robot

6.20. Pruebas de aceptación en entornos complejos simulados 3

Para validar el rendimiento del software en escenarios más desafiantes, se llevaron a cabo pruebas de aceptación en entornos complejos simulados. En estas pruebas, se introdujeron obstáculos adicionales y laberintos más elaborados en el lienzo de simulación, lo que permitió observar el comportamiento del algoritmo ante situaciones de alta densidad de barreras. Los resultados mostraron que el algoritmo es capaz de adaptarse a estos entornos, generando rutas eficientes incluso con una mayor presencia de obstáculos. Sin embargo, se identificaron áreas donde el tiempo de cálculo aumenta significativamente en mapas de gran tamaño, lo que sugiere la necesidad de optimizar ciertos componentes del algoritmo en futuras iteraciones

6.20.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión $n \times n$ correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código.
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.20.2. 2da Prueba de aceptación: Carga de mapas

Descripción: En el código se cargan imágenes, las cuales corresponden a mapas, donde éstas se cargan para colocarlos en el lienzo al iniciar el programa

Flujo:

- Se coloca la dirección de la imagen
- Se coloca un tamaño para tanto la imagen como para el lienzo
- Se inicia el programa.

- El programa al iniciar, carga la configuración, mostrando la imagen del mapa, convertido al lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.
- La imagen del mapa es colocada correctamente en el lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.20.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado. Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.20.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.

- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.20.5. 5ta Prueba de aceptación: Inicio de la simulación

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

Estas pruebas fueron realizadas en entornos simulados de tamaño más grande, y los resultados revelaron que las rutas generadas continúan siendo precisas, aunque con un mayor tiempo de procesamiento, especialmente en sistemas con recursos limitados.

6.21. Evaluación final de desempeño del software 3

Para la evaluación final del simulador, esta vez no fue necesario del manejo en espacios mucho más grandes en cuanto al tamaño del lienzo, sino a entornos más acordes a la realidad donde hay más posibilidad de encontrar algún fallo o generación de rutas u obstáculos fuera de lo originalmente planteado. Un ejemplo es la generación de una ruta para llegar de un punto a otro dentro del circuito del Instituto Politécnico Nacional, en el cual se puso a prueba el simulador

para poder encontrar la ruta dentro de una locación a otra en este circuito y comprobar cómo es que el simulador se comportaba uniendo tanto el cargado de mapas, como de la generación de las rutas a través del Physarum.

En las Figuras 84 y 85, se muestran algunos de los datos recopilados con los que fue evaluado el software en la generación de rutas.

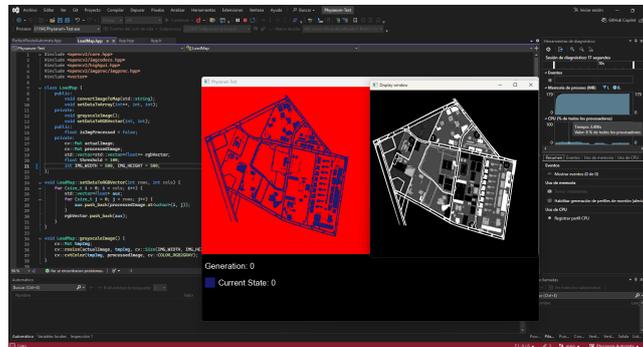


Figura 84: Análisis del simulador sobre el porcentaje de uso del procesador.

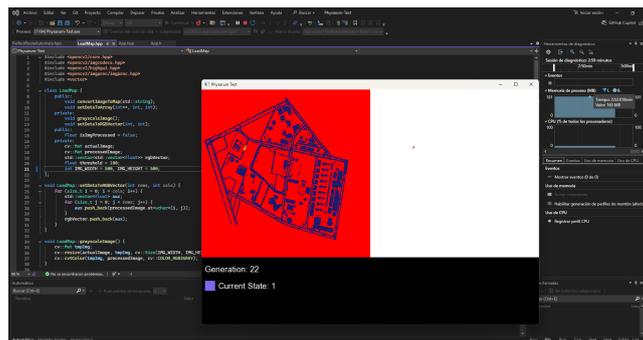


Figura 85: Análisis del uso de memoria por parte del simulador durante la ejecución del programa en la generación de rutas.

Con la información recopilada, podemos darnos cuenta de que mientras más grande el lienzo y la imagen que se le sea proporcionada al simulador, el programa tiene un uso mayor del procesador, así como del uso de la memoria debido a que en ese momento hay muchos más píxeles que dibujar en pantalla, así como también el tamaño del arreglo se vuelve mucho más grande, lo que hace complicado y más tardado la evaluación y la generación de la ruta debido a que no se está lo suficientemente optimizado para espacios mucho más grandes.

6.22. Recopilación final de datos del software para el informe 3

Se mapeó una parte correspondiente a los edificios pertenecientes a la Escuela Superior de Cómputo, perteneciente al Instituto Politécnico Nacional, en el cual se generó una ruta la cual el robot debía de seguir para poder llegar desde un salón a otro. Esta prueba de crucial, debido a que es realizada en un entorno que, a pesar de que en el mapa no es del todo claro, representa un reto debido a que en la vida real hay mucha gente la cual pasará cerca del robot y podría implicar cierto riesgo a la hora de que el robot realice su trayecto, pudiendo llegar a ocasionar un accidente o posiblemente un fallo debido a la posible interferencia.

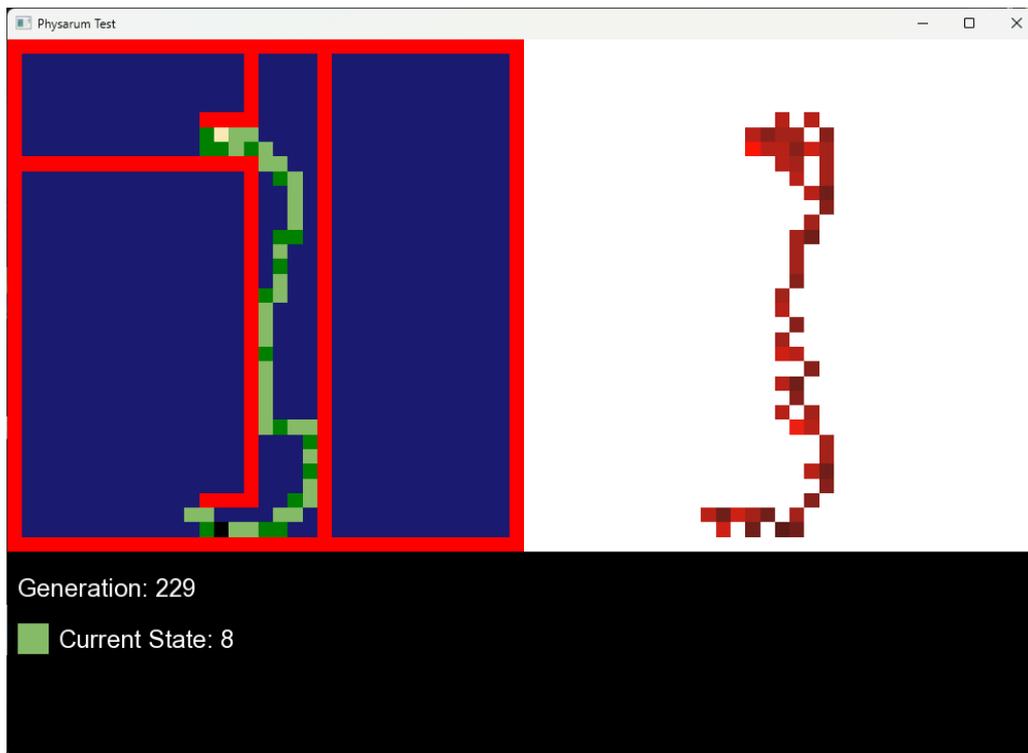


Figura 86: Esta es la ruta generada la cual el robot debe de seguir para poder llegar desde un salón hasta el otro. La célula negra representa el estado inicial y la célula amarilla representa el destino final (nutriente encontrado).

Como se puede observar en la Figura 86, la ruta generada por el algoritmo es la que se muestra en color verde, la cual es la ruta que el robot deberá de seguir para poder llegar desde el salón 1 al salón 2. Una vez generada la ruta, sus coordenadas fueron almacenadas y enviadas hacia el robot, el cual se encargó de interpretarlas y así seguir la ruta correspondiente a el mapa donde primeramente fue iniciada la simulación del Physarum. Esta ruta fue parcialmente corregida debido a las implicaciones que tiene el comportamiento natural del Physarum, en el cual, la ruta encontrada no necesariamente es de las mejores que se pueden obtener, debido al comportamiento del plasmodio que tiene de manera intrínseca

Como se puede observar en las Figuras 87 - 89, el robot sigue la ruta entre los dos salones, mostrando así cómo es que el algoritmo ha generado la ruta completamente, así también mostrando el comportamiento del robot mientras sigue la ruta, en el cual constantemente se esta monitoreando para evitar cualquier incidente con las personas que recorren el lugar.

Así podemos decir que las pruebas realizadas fueron satisfactorias, y los datos recopilados muestran que el comportamiento del robot es el esperado, por lo que esta y más información nos representan una gran área de mejora para los respectivos casos en los cuales podamos mapear diferentes tipos de lugares y locaciones con incluso más información la cual pueda representar un mayor reto para el robot.



Figura 87: Robot saliendo del salón 1 y siguiendo la ruta generada por el algoritmo del Physarum.



Figura 88: Robot a mitad de camino siguiendo la ruta generada por el algoritmo del Physarum.

6.23. Análisis de primera iteración de datos de sensores

En esta primera iteración, se recopilaron y analizaron los datos de los sensores integrados en el sistema, con el objetivo de comprender el tipo de información proporcionada por el LiDAR y las cámaras utilizadas, así como su potencial para la implementación de una interfaz gráfica. Esta fase fue crucial, ya que marcó el punto de partida para la integración de los datos sensoriales en un entorno visual que permitiera la interpretación y el control en tiempo real.

El sensor LiDAR proporcionó datos en formato de coordenadas cartesianas, calculadas a partir de distancias y ángulos. Estos datos permiten la representación de un mapa del entorno en dos dimensiones, detectando la proximidad de objetos y obstáculos en el área circundante. La información recopilada incluyó principalmente mediciones de distancia en milímetros, asociadas a coordenadas angulares específicas, las cuales se convirtieron en valores cartesianas para su futura visualización en una interfaz gráfica. Por ejemplo, las primeras mediciones capturadas



Figura 89: Robot llegando al destino final siguiendo la ruta generada por el algoritmo del Physarum.

del LiDAR ofrecían distancias que oscilaban entre los 20 cm y los 5 metros, brindando una imagen básica del espacio inmediato.

Por otro lado, se evaluaron diferentes tipos de cámaras, las cuales proporcionaban datos en diversos formatos. Se analizaron cámaras que ofrecían salidas en formatos como Modelo de color YUV (no tiene un significado literal en términos de siglas, pero comúnmente se describe por sus componentes: Y para luminancia, y U y V para crominancia), incluyendo variantes (YUV420, YUV422) y RGB, siendo el formato YUV uno de los más eficientes para transmisión y procesamiento de video, especialmente en aplicaciones que requieren una compresión de alta calidad. Este formato divide la información de luminancia (Y) y crominancia (UV), lo que facilita la reducción del ancho de banda necesario sin comprometer la calidad visual en exceso. Durante las pruebas iniciales, el formato YUV422 mostró ser particularmente adecuado para entornos donde la transmisión rápida de imágenes era prioritaria, dado que proporcionaba una mayor eficiencia en términos de compresión de datos en comparación con RGB, aunque con menos fidelidad en los detalles de color.

La cámara RGB también fue evaluada para obtener imágenes con una representación precisa de los colores del entorno. A diferencia del formato YUV, los datos RGB capturan la totalidad de la información de color, lo que resultó útil en situaciones donde se requería una mayor claridad visual para la identificación de objetos. Sin embargo, el formato RGB mostró ser menos eficiente en términos de tamaño de archivo y procesamiento, lo que lo hacía menos adecuado para aplicaciones en tiempo real donde la optimización del rendimiento era crítica.

El análisis de esta primera iteración fue fundamental para comprender las capacidades y limitaciones de los sensores, lo que permitió decidir sobre los formatos de datos más adecuados para su futura implementación en una interfaz gráfica. Se identificó que los datos del LiDAR ofrecían una representación clara y precisa

del entorno inmediato, mientras que las diferentes cámaras evaluadas proporcionaban flexibilidad en la captura de información visual, dependiendo de los requisitos de resolución y velocidad de procesamiento.

6.24. Diseño de módulos de hardware del robot para pruebas de integración

En esta sección se detalla el diseño de los módulos de hardware del robot que hemos desarrollado para el Trabajo Terminal (TT). El objetivo principal es asegurar una integración eficiente entre los diferentes componentes, permitiendo así la realización de pruebas de integración efectivas. La modularidad del diseño facilita tanto el desarrollo como el mantenimiento del sistema, así como también la sustitución de algunos componentes, esto nos permite aislar y solucionar problemas de manera aislada.

Nuestro robot desarrollado consta de cuatro módulos principales: actuadores, sensores, unidad de control y módulo de visualización. Cada uno de estos módulos cumple una función específica que permite al robot moverse, detectar obstáculos, y seguir rutas predefinidas. En la Figura 90 se muestra el diseño de los pines de configuración de los módulos de hardware del robot.

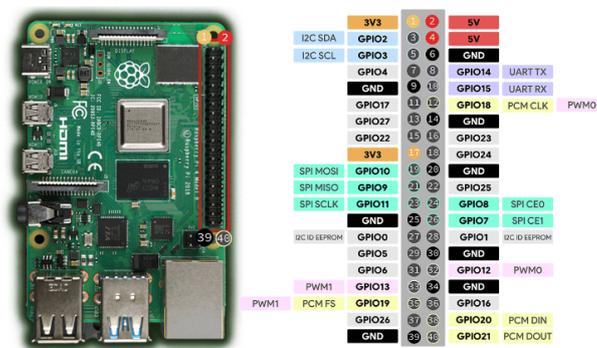


Figura 90: Diseño de los pines de configuración de los módulos de hardware del robot.

6.24.1. Módulo de actuadores (Motores)

El robot está equipado por cuatro motores a pasos Nema 23, controlados a través de un controlador driver específico para motores a pasos. Estos motores son responsables del movimiento del robot en las direcciones deseadas, permitiendo giros, avances y retrocesos mediante la manipulación precisa de los pines de control Modulación por Ancho de Pulso (Pulse Width Modulation, PWM) y de dirección.

Los motores Nema 23 fueron elegidos debido a su precisión en el posicionamiento y su capacidad para generar el torque necesario para mover la estructura del robot, que está construida con perfiles de aluminio y ruedas omnidireccionales. En el Cuadro 9 se muestran las especificaciones de los motores a pasos Nema 23.

Producto	Descripción	Piezas
Motor a pasos Nema 23	Motor a pasos Nema 23 con placa frontal	4
Controlador de motores a pasos	Controlador de motores a pasos TB6600	4
Rueda omnidireccional	Rueda omnidireccional de 6 in	4

Cuadro 9: Especificaciones de los motores a pasos Nema 23.

Los motores están conectados a los pines Entrada/Salida de Propósito General (General Purpose Input/Output, GPIO) de la Raspberry Pi 4 B mediante un controlador de motores. El sistema de control utiliza señales PWM para regular la velocidad de los motores, y señales de dirección para controlar el sentido de giro.

Elegimos los Motores Nema 23 debido a su precisión y capacidad de generar torque necesario para mover el robot con su estructura de aluminio y ruedas omnidireccionales. Su tamaño y especificaciones son ideales para aplicaciones que requieren alta precisión en el movimiento.

La incorporación de ruedas omnidireccionales de 6 pulgadas permite al robot realizar movimientos en varias direcciones sin necesidad de girar su estructura por completo. Esto optimiza la maniobrabilidad en entornos reducidos

Finalmente elegimos los controladores a pasos por que estos son capaces de manejar la corriente y el voltaje necesario para los motores Nema 23, permitiendo un control preciso mediante señales PWM generadas desde la unidad de control.

6.24.2. Módulo de sensores (LiDAR)

El robot está equipado por un sensor YLiDAR 2XL, que permite realizar un escaneo de 360° del entorno para detectar obstáculos y mapear el área de operación del robot. La elección de este sensor se debe a su capacidad para medir distancias con alta precisión y su adecuado rango de operación, lo que lo convierte en una opción ideal para la navegación autónoma en tiempo real. El sensor se conecta a la unidad de control mediante un puerto USB tipo A.

El sensor se conecta a la unidad de control mediante un puerto USB tipo A.

El sensor de distancia se eligió por su capacidad de ofrecer un rango de detección de 8 metros de diámetro y su alta precisión en el mapeo de entornos. Su ángulo de escaneo de 360° permite que el robot detecte obstáculos en cualquier dirección.

Esta subsección no tiene un esquema de conexión por que simplemente se conecta por USB tipo A.

6.24.3. Unidad de control (Raspberry Pi 4 B)

La unidad de Control es el cerebro del robot, responsable de gestionar y coordinar todos los módulos de hardware, incluyendo los actuadores y los sensores. En este proyecto, se utiliza una Raspberry Pi 4 B, que ofrece la capacidad de procesamiento y la conectividad necesarias para integrar los diferentes componentes del sistema.

La Raspberry Pi controla los motores mediante GPIO, recibe los datos del LiDAR a través del puerto Transmisor-Receptor Asíncrono Universal (Universal Asynchronous Receiver-Transmitter, UART), y además se encarga de la visualización de la información mediante una pantalla conectada o mediante el protocolo VNC.

El diseño del módulo de control se basa en la capacidad de la Raspberry Pi de gestionar múltiples tareas simultáneamente, coordinando el movimiento del robot, la recepción de datos de sensores, y la visualización de la información en una pantalla conectada.

Se eligió la Raspberry Pi 4 B debido a su capacidad para manejar múltiples interfaces de

comunicación (GPIO, UART) y su potencia de procesamiento, lo cual es crucial para la gestión de múltiples módulos de hardware en tiempo real.

6.24.4. Módulo de visualización

El módulo de visualización permite monitorizar el entorno y la posición del robot en tiempo real. Para ello, se utiliza una cámara de visión nocturna conectada a la Raspberry Pi y un sistema de visualización que integra los datos del sensor LiDAR en un mapa gráfico. La visualización se realiza mediante una interfaz gráfica que muestra tanto la imagen capturada por la cámara como el mapa generado por los datos del LiDAR.

Elegimos una cámara infrarroja nocturna de 5MP para capturar video del entorno y verlo en la interfaz gráfica.

El módulo de visualización está diseñado para combinar la salida de la cámara con la representación gráfica de los datos del LiDAR, lo que facilita el monitoreo del entorno en tiempo real y la toma de decisiones basadas en la información recopilada. Se puede ver un ejemplo de la interfaz gráfica en la Figura 91.



Figura 91: Visualización de la interfaz gráfica del robot.

La cámara infrarroja nocturna se eligió por su capacidad para capturar imágenes de alta calidad en condiciones de poca luz, lo que permite al robot operar en entornos con iluminación deficiente.

6.24.5. Pruebas de integración

Como parte del proceso de validación de los módulos de hardware, se han planificado pruebas de integración cuyo objetivo será asegurar que todos los módulos (actuadores, sensores, unidad de control y visualización) funcionen de manera conjunta y coordinada. Estas pruebas estarán diseñadas para verificar que los módulos puedan interactuar de forma efectiva en diversos escenarios, garantizando que el robot cumpla con los requisitos de funcionamiento autónomo.

Las pruebas de integración se dividirán en varias fases, las cuales serán las siguientes:

A) Prueba de Conectividad entre módulos

- Objetivo: Verificar que los módulos de hardware se pueden comunicar entre sí de manera efectiva.
- Pasos:

- a) Configurar la conexión UART entre la Raspberry Pi 4 B y el LiDAR YLIDAR 2XL.
- b) Verificar la comunicación entre la Raspberry Pi 4 B y los motores a través de los pines GPIO (señales PWM y dirección).
- c) Comprobar la conexión de la cámara a la Raspberry Pi 4B.
- Métrica de éxito: El tiempo de respuesta entre la señal enviada desde la Raspberry Pi 4 B y la recepción de datos en los actuadores o sensores debe de ser menor a 100 ms.
- Criterio de éxito: La conexión entre los módulos debe mantenerse estable durante al menos 1 hora de operación continua, sin pérdida de datos ni interrupciones.

B) Pruebas de movilidad y control de actuadores

- Objetivo: Evaluar la capacidad del robot para ejecutar movimientos básicos (avance, retroceso, giros) de manera precisa y controlada.
- Pasos:
 - a) Enviar señales desde la unidad de control a los motores para probar el avance, retroceso y giros.
 - b) Ajustar la frecuencia PWM para probar diferentes velocidades de movimiento.
 - c) Medir la precisión del movimiento en distancias cortas y largas.
- Métrica de éxito: Desviación máxima de 5cm en distancias cortas y de 10 cm en distancias largas.
- Criterios de éxito: Los motores deben responder en menos de 500 ms tras recibir las señales de control y realizar los movimientos sin errores.

C) Pruebas de detección de obstáculos

- Objetivo: Verificar la capacidad del robot para detectar obstáculos en su entorno y reaccionar de manera adecuada para evitar colisiones.
- Pasos:
 - a) Realizar un escaneo completo del entorno con el sensor LiDAR.
 - b) Analizar los datos del LiDAR para identificar obstáculos y calcular las distancias de seguridad.
 - c) Enviar señales de control a los motores para evitar los obstáculos detectados.
- Métrica de éxito: El robot debe ser capaz de detectar obstáculos a una distancia mínima de 15cm y reaccionar de manera efectiva para evitar colisiones.
- Criterios de éxito: El robot debe ser capaz de evitar obstáculos en un entorno cerrado con una tasa de éxito del 90 %.

D) Pruebas de Seguimiento de Rutas Predefinidas

- Objetivo: Evaluar la capacidad del robot para seguir rutas prediseñadas, ajustando su trayectoria en respuesta a cambios en el entorno.
- Pasos:
 - a) Trazar una ruta en un entorno controlado.
 - b) Activar el sistema de seguimiento de rutas del robot.

- c) Introducir obstáculos inesperados y verificar que el robot ajuste su trayectoria sin desviarse significativamente.
- Métrica de éxito: El robot debe seguir la ruta trazada con una desviación máxima de 10 cm en trayectorias rectas y de 15 cm en giros.
- Criterio de éxito: El robot debe ajustar su trayectoria en menos de 1 segundo tras detectar un cambio en el entorno.

E) Pruebas de Visualización en Tiempo Real

- Objetivo: Verificar que la interfaz gráfica del robot muestra la información del entorno y la posición del robot en tiempo real.
- Pasos:
 - a) Iniciar la cámara de visión nocturna y el sensor LiDAR.
 - b) Mostrar la imagen capturada por la cámara y el mapa generado por el LiDAR en la interfaz gráfica.
 - c) Verificar que la información se actualiza en tiempo real y que el robot se representa correctamente en el mapa.
- Métrica de éxito: La interfaz gráfica debe mostrar la información del entorno y la posición del robot con una latencia máxima de 500 ms.
- Criterio de éxito: La interfaz gráfica debe actualizarse en tiempo real y mostrar la información del entorno y la posición del robot de manera precisa.

6.25. Desarrollo de módulos de hardware del robot para pruebas de integración

En la siguiente sección vamos a detallar el proceso de implementación de los módulos de hardware que componen el robot. Se describirán las decisiones técnicas tomadas durante la construcción, la integración física y funcional de los módulos, y los resultados preliminares de las pruebas iniciales. El objetivo es mostrar cómo los módulos de actuadores, sensores, la unidad de control y sistema de visualización fueron desarrollados para llevar a cabo las pruebas de integración.

6.25.1. Módulo de actuadores

El módulo de actuadores está compuesto por cuatro motores Nema 23 conectados a ruedas omnidireccionales. Los motores se montaron sobre una base de perfiles de aluminio 2040, elegidos por su resistencia y ligereza. Los motores se conectaron a sus respectivos controladores mediante cables blindados para reducir interferencias electromagnéticas.

Para el control de los motores, se utilizó la Raspberry Pi, que envía señales Modulación por Ancho de Pulso (Pulse Width Modulation, PWM) a través de sus pines Entrada/Salida de Propósito General (General Purpose Input/Output, GPIO), controlando la velocidad y dirección de los motores. Se ajustó la frecuencia de las señales PWM a 400 Hz, lo que permitió un control suave y preciso del movimiento del robot.

Para poder controlar los motores, se desarrolló un código en C++ que utilizó la biblioteca pigpio para gestionar las señales PWM. El código permite ajustar tanto la velocidad como la dirección de los motores en tiempo real, se muestra un ejemplo en el Listing 11.

Listing 11: Código de ejemplo de motores

```
1     void setMotorSpeed(int motor, int frequency) {
2         if (motor >= 0 && motor < 4) {
3             gpioSetPWmfrequency(PWM_PINS[motor], frequency);
4         }
5     }
6     void setMotorDirection(int motor, int direction) {
7         if (motor >= 0 && motor < 4) {
8             gpioWrite(DIR_PINS[motor], direction);
9         }
10    }
```

6.25.2. Módulo de sensores

El sensor LiDAR YDLIDAR 2XL fue montado en la parte superior del robot para obtener un ángulo de escaneo de 360 grados. El sensor se conectó a la Raspberry Pi a través de un puerto USB A. La configuración física permitió que el LiDAR tuviera una vista despejada del entorno, crucial para la detección precisa de obstáculos.

El sensor LiDAR se configuró utilizando la biblioteca proporcionada por el fabricante (YLiDAR SDK) y se desarrolló un código para la lectura continua de los datos de escaneo. Los datos obtenidos del LiDAR se procesaron en tiempo real para permitir la navegación autónoma y la detección de obstáculos, en la Listing 12 se muestra un ejemplo de código.

Listing 12: Primero se implementa la librería del LiDAR

```
1     #include "CYdLidar.h"
2     using namespace ydlidar;
```

Después se inicializa el sensor y se obtienen los datos de escaneo, se muestra un ejemplo en la Listing 13.

Listing 13: Configuración de opciones del LiDAR

```
1     CYdLidar laser;
2     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
3     std::string ignore_array;
4     ignore_array.clear();
5     laser.setlidaropt(LidarPropIgnoreArray, ignore_array.c_str(),
6         ignore_array.size());
7
8     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int))
9     ;
10    int optval = TYPE_TRIANGLE;
11    laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
12    optval = YDLIDAR_TYPE_SERIAL;
13    laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
14    optval = isSingleChannel ? 3 : 4;
15    laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
16    optval = 4;
17    laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof(int));
18
19    bool b_optvalue = false;
```

```

18     laser.setlidaropt(LidarPropFixedResolution, &b_optvalue, sizeof(
        bool));
19     laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool));
20     laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool));
21     b_optvalue = true;
22     laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(bool
        ));
23     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof
        (bool));
24     b_optvalue = false;
25     laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool));
26     b_optvalue = true;
27     laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
        sizeof(bool));
28     b_optvalue = false;
29     laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue, sizeof(
        bool));
30
31     float f_optvalue = 180.0f;
32     laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float));
33     f_optvalue = -180.0f;
34     laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float));
35     f_optvalue = 64.0f;
36     laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float));
37     f_optvalue = 0.05f;
38     laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float));
39     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float
        ));

```

6.25.3. Unidad de control

La Raspberry Pi 4 B fue montada en el centro del robot para minimizar la longitud de los cables de conexión hacia los motores, el LiDAR y la cámara. Se utilizó una estructura de soporte para asegurar la Raspberry Pi en su lugar, proporcionando un acceso fácil a sus puertos GPIO y Transmisor-Receptor Asíncrono Universal (Universal Asynchronous Receiver-Transmitter, UART).

Se desarrolló un sistema de control centralizado en la Raspberry Pi, utilizando programación en C++ para gestionar las señales enviadas a los actuadores y procesarlos datos de los sensores. Este sistema permite la toma de decisiones en tiempo real, como ajustar la trayectoria del robot según los datos del LiDAR. Esto se ve en la Listing 14.

Listing 14: Código de ejemplo de Kinect y LiDAR

```

1     #include "CYdLidar.h"
2     #include <string>
3     #include <map>
4     #include <iostream>
5     #include <SFML/Graphics.hpp>
6     #include <opencv2/opencv.hpp>
7     #include <libfreenect.hpp>
8     #include <thread>
9     #include <atomic>
10    #include <vector>
11
12    using namespace std;
13    using namespace ydlidar;

```

```

14     using namespace cv;
15
16     #if defined(_MSC_VER)
17     #pragma comment(lib, "ydlidar_sdk.lib")
18     #endif
19
20     // Freenect variables
21     freenect_context *f_ctx;
22     freenect_device *f_dev;
23     Mat depthMat(Size(640, 480), CV_16UC1);
24     Mat rgbMat(Size(640, 480), CV_8UC4, Scalar(0, 0, 0, 0)); //
25     Initialize with transparent
26     atomic<bool> is_running(true);
27
28     void depth_cb(freenect_device *dev, void *depth, uint32_t
29     timestamp) {
30         Mat depth_tmp(Size(640, 480), CV_16UC1, depth);
31         depth_tmp.copyTo(depthMat);
32         cout << "Depth frame received at timestamp: " << timestamp <<
33         endl;
34     }
35
36     void rgb_cb(freenect_device *dev, void *rgb, uint32_t timestamp) {
37         Mat rgb_tmp(Size(640, 480), CV_8UC3, rgb);
38         cvtColor(rgb_tmp, rgbMat, COLOR_RGB2RGBA);
39         cout << "RGB frame received at timestamp: " << timestamp <<
40         endl;
41     }
42
43     void drawGrid(sf::RenderTarget &target, float gridSpacing, float
44     offsetX, float offsetY) {
45         sf::Color gridColor(200, 200, 200); // Light gray color
46         for (float x = offsetX; x < target.getSize().x; x +=
47         gridSpacing) {
48             sf::Vertex line[] = {
49                 sf::Vertex(sf::Vector2f(x, 0), gridColor),
50                 sf::Vertex(sf::Vector2f(x, target.getSize().y),
51                 gridColor)
52             };
53             target.draw(line, 2, sf::Lines);
54         }
55         for (float x = offsetX; x > 0; x -= gridSpacing) {
56             sf::Vertex line[] = {
57                 sf::Vertex(sf::Vector2f(x, 0), gridColor),
58                 sf::Vertex(sf::Vector2f(x, target.getSize().y),
59                 gridColor)
60             };
61             target.draw(line, 2, sf::Lines);
62         }
63         for (float y = offsetY; y < target.getSize().y; y +=
64         gridSpacing) {
65             sf::Vertex line[] = {
66                 sf::Vertex(sf::Vector2f(0, y), gridColor),
67                 sf::Vertex(sf::Vector2f(target.getSize().x, y),
68                 gridColor)
69             };
70             target.draw(line, 2, sf::Lines);
71         }
72         for (float y = offsetY; y > 0; y -= gridSpacing) {
73             sf::Vertex line[] = {

```

```

64         sf::Vertex(sf::Vector2f(0, y), gridColor),
65         sf::Vertex(sf::Vector2f(target.getSize().x, y),
66                     gridColor)
67     };
68     target.draw(line, 2, sf::Lines);
69 }
70
71 sf::Color getPointColor(float distance, float maxRange) {
72     float ratio = distance / maxRange;
73     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
74     from red to green
75 }
76
77 void drawNorthIndicator(sf::RenderTexture &lidarTexture, float
78     offsetX, float offsetY) {
79     sf::Color northColor(255, 0, 0); // Red color for the north
80     indicator
81     float length = 50.0f; // Length of the north indicator line
82
83     sf::Vertex line[] = {
84         sf::Vertex(sf::Vector2f(offsetX, offsetY), northColor),
85         sf::Vertex(sf::Vector2f(offsetX, offsetY - length),
86                     northColor) // Line pointing upwards
87     };
88     lidarTexture.draw(line, 2, sf::Lines);
89 }
90
91 void detectAndDrawDepthObstacles(Mat &depthMat, sf::RenderTexture
92     &lidarTexture, float offsetX, float offsetY, float scale,
93     vector<Point2f> &depthObstacles) {
94     for (int y = 0; y < depthMat.rows; y += 10) {
95         for (int x = 0; x < depthMat.cols; x += 10) {
96             uint16_t depth_value = depthMat.at<uint16_t>(y, x);
97             if (depth_value > 0 && depth_value < 2047) {
98                 float depth_in_meters = depth_value / 1000.0f;
99
100                 float adjustedX = offsetX + (x - depthMat.cols /
101                     2) * scale / depth_in_meters;
102                 float adjustedY = offsetY - (y - depthMat.rows /
103                     2) * scale / depth_in_meters; // Invert Y to
104                 match screen coordinates
105
106                 sf::CircleShape circle(3); // Small circle to
107                 represent the obstacle
108                 circle.setPosition(adjustedX, adjustedY);
109                 circle.setFillColor(sf::Color::Red);
110
111                 lidarTexture.draw(circle);
112
113                 // Add the obstacle to the vector
114                 depthObstacles.push_back(Point2f(adjustedX,
115                     adjustedY));
116             }
117         }
118     }
119 }
120
121 void compareObstacles(const vector<Point2f> &depthObstacles, const
122     vector<Point2f> &lidarObstacles) {

```

```

111     cout << "Comparing Obstacles:" << endl;
112     for (const auto &d : depthObstacles) {
113         bool matchFound = false;
114         for (const auto &l : lidarObstacles) {
115             float distance = sqrt(pow(d.x - l.x, 2) + pow(d.y - l.
116                 y, 2));
117             if (distance < 10.0f) { // If the distance is less
118                 than a threshold, consider it a match
119                 matchFound = true;
120                 break;
121             }
122         }
123         if (matchFound) {
124             cout << "Match found for depth obstacle at (" << d.x
125                 << ", " << d.y << ")" << endl;
126         } else {
127             cout << "No match for depth obstacle at (" << d.x << "
128                 , " << d.y << ")" << endl;
129         }
130     }
131 }
132
133 void freenect_thread_func() {
134     while (is_running) {
135         freenect_process_events(f_ctx);
136     }
137 }
138
139 void print_frame_modes() {
140     cout << "Available video modes:" << endl;
141     for (int res = FREENECT_RESOLUTION_LOW; res <=
142         FREENECT_RESOLUTION_HIGH; ++res) {
143         for (int fmt = FREENECT_VIDEO_RGB; fmt <=
144             FREENECT_VIDEO_IR_8BIT; ++fmt) {
145             freenect_frame_mode mode = freenect_find_video_mode((
146                 freenect_resolution)res, (freenect_video_format)
147                 fmt);
148             if (mode.is_valid) {
149                 cout << "Resolution: " << res << ", Format: " <<
150                     fmt << ", Width: " << mode.width << ", Height:
151                     " << mode.height << ", Bytes per pixel: " <<
152                     mode.data_bits_per_pixel << endl;
153             }
154         }
155     }
156
157     cout << "Available depth modes:" << endl;
158     for (int res = FREENECT_RESOLUTION_LOW; res <=
159         FREENECT_RESOLUTION_HIGH; ++res) {
160         for (int fmt = FREENECT_DEPTH_11BIT; fmt <=
161             FREENECT_DEPTH_REGISTERED; ++fmt) {
162             freenect_frame_mode mode = freenect_find_depth_mode((
163                 freenect_resolution)res, (freenect_depth_format)
164                 fmt);
165             if (mode.is_valid) {
166                 cout << "Resolution: " << res << ", Format: " <<
167                     fmt << ", Width: " << mode.width << ", Height:
168                     " << mode.height << ", Bytes per pixel: " <<
169                     mode.data_bits_per_pixel << endl;
170             }
171         }
172     }
173 }

```

```

153     }
154 }
155 }
156
157 int main(int argc, char *argv[]) {
158     std::string port;
159     ydlidar::os_init();
160
161     // Initialize Freenect
162     if (freenect_init(&f_ctx, NULL) < 0) {
163         cerr << "Freenect init failed" << endl;
164         return -1;
165     }
166     freenect_set_log_level(f_ctx, FREENECT_LOG_DEBUG);
167     int nr_devices = freenect_num_devices(f_ctx);
168     if (nr_devices < 1) {
169         cerr << "No Kinect devices found" << endl;
170         return -1;
171     }
172     if (freenect_open_device(f_ctx, &f_dev, 0) < 0) {
173         cerr << "Could not open Kinect device" << endl;
174         return -1;
175     }
176
177     // Print available frame modes
178     print_frame_modes();
179
180     // Set the video and depth modes
181     freenect_frame_mode video_mode = freenect_find_video_mode(
182         FREENECT_RESOLUTION_MEDIUM, FREENECT_VIDEO_RGB);
183     if (!video_mode.is_valid) {
184         cerr << "Invalid video mode" << endl;
185         return -1;
186     }
187     freenect_frame_mode depth_mode = freenect_find_depth_mode(
188         FREENECT_RESOLUTION_MEDIUM, FREENECT_DEPTH_11BIT);
189     if (!depth_mode.is_valid) {
190         cerr << "Invalid depth mode" << endl;
191         return -1;
192     }
193     if (freenect_set_video_mode(f_dev, video_mode) < 0) {
194         cerr << "Failed to set video mode" << endl;
195         return -1;
196     }
197     if (freenect_set_depth_mode(f_dev, depth_mode) < 0) {
198         cerr << "Failed to set depth mode" << endl;
199         return -1;
200     }
201
202     freenect_set_depth_callback(f_dev, depth_cb);
203     freenect_set_video_callback(f_dev, rgb_cb);
204     freenect_start_depth(f_dev);
205     freenect_start_video(f_dev);
206
207     // Start Kinect processing thread
208     std::thread freenect_thread(freenect_thread_func);
209
210     // Get available LiDAR ports

```

```

211     std::map<std::string, std::string> ports = ydlidar::
        lidarPortList();
212     if (ports.size() == 1) {
213         port = ports.begin()->second;
214     } else {
215         int id = 0;
216         for (auto it = ports.begin(); it != ports.end(); it++) {
217             printf("%d. %s\n", id, it->first.c_str());
218             id++;
219         }
220         if (ports.empty()) {
221             printf("No LiDAR was detected. Please enter the LiDAR
                serial port:");
222             std::cin >> port;
223         } else {
224             while (ydlidar::os_isOk()) {
225                 printf("Please select the LiDAR port:");
226                 std::string number;
227                 std::cin >> number;
228                 if ((size_t)atoi(number.c_str()) >= ports.size())
                continue;
229                 auto it = ports.begin();
230                 id = atoi(number.c_str());
231                 while (id) {
232                     id--;
233                     it++;
234                 }
235                 port = it->second;
236                 break;
237             }
238         }
239     }
240
241     // Baud rate selection
242     int baudrate = 115200;
243     printf("Baudrate: %d\n", baudrate);
244     if (!ydlidar::os_isOk()) {
245         return 0;
246     }
247
248     // Check for single channel communication
249     bool isSingleChannel = false;
250     std::string input_channel;
251     isSingleChannel = true;
252
253     if (!ydlidar::os_isOk()) {
254         return 0;
255     }
256
257     // Scan frequency
258     float frequency = 8.0;
259     while (ydlidar::os_isOk() && !isSingleChannel) {
260         printf("Please enter the LiDAR scan frequency [5-12]:");
261         std::string input_frequency;
262         std::cin >> input_frequency;
263         frequency = atof(input_frequency.c_str());
264         if (frequency <= 12 && frequency >= 5.0) {
265             break;
266         }
267         fprintf(stderr, "Invalid scan frequency. The scanning

```

```

        frequency range is 5 to 12 Hz. Please re-enter.\n");
268     }
269
270     if (!ydlidar::os_isOk()) {
271         return 0;
272     }
273
274     CYdLidar laser;
275     //////////////////////////////////string property////////////////////////////////
276     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
        ());
277     std::string ignore_array;
278     ignore_array.clear();
279     laser.setlidaropt(LidarPropIgnoreArray, ignore_array.c_str(),
        ignore_array.size());
280
281     //////////////////////////////////int property////////////////////////////////
282     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
        int));
283     int optval = TYPE_TRIANGLE;
284     laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
285     optval = YDLIDAR_TYPE_SERIAL;
286     laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
287     optval = isSingleChannel ? 3 : 4;
288     laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
289     optval = 4;
290     laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof
        (int));
291
292     //////////////////////////////////bool property////////////////////////////////
293     bool b_optvalue = false;
294     laser.setlidaropt(LidarPropFixedResolution, &b_optvalue,
        sizeof(bool));
295     laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool
        ));
296     laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool
        ));
297     b_optvalue = true;
298     laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(
        bool));
299     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
        sizeof(bool));
300     b_optvalue = false;
301     laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool
        ));
302     b_optvalue = true;
303     laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
        sizeof(bool));
304     b_optvalue = false;
305     laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue,
        sizeof(bool));
306
307     //////////////////////////////////float property////////////////////////////////
308     float f_optvalue = 180.0f;
309     laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float
        ));
310     f_optvalue = -180.0f;
311     laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float
        ));
312     f_optvalue = 64.0f;

```

```

313     laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float
        ));
314     f_optvalue = 0.05f;
315     laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float
        ));
316     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
        float));
317
318     // Initialize LiDAR
319     bool ret = laser.initialize();
320     if (ret) {
321         // Start scanning
322         ret = laser.turnOn();
323     } else {
324         cerr << "Error initializing YDLIDAR: " << laser.
            DescribeError() << endl;
325         return -1;
326     }
327
328     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera and
        LiDAR Visualization");
329
330     // Create SFML texture and sprite for the camera feed
331     sf::Texture cameraTexture;
332     sf::Sprite cameraSprite;
333
334     // Main loop
335     while (ret && window.isOpen() && ydlidar::os_isOk()) {
336         sf::Event event;
337         while (window.pollEvent(event)) {
338             if (event.type == sf::Event::Closed)
339                 window.close();
340         }
341
342         LaserScan scan;
343
344         if (laser.doProcessSimple(scan)) {
345             window.clear();
346
347             // Update the camera texture with the frame data
348             if (!cameraTexture.create(rgbMat.cols, rgbMat.rows)) {
349                 cerr << "Failed to create texture" << endl;
350                 break;
351             }
352             cameraTexture.update(rgbMat.ptr());
353
354             cameraSprite.setTexture(cameraTexture);
355             cameraSprite.setScale(
356                 window.getSize().x / static_cast<float>(
                    cameraTexture.getSize().x),
357                 window.getSize().y / static_cast<float>(
                    cameraTexture.getSize().y)
358             );
359
360             // Draw the camera feed
361             window.draw(cameraSprite);
362
363             // Draw the LiDAR minimap
364             sf::RenderTexture lidarTexture;
365             lidarTexture.create(300, 300);

```

```

366     lidarTexture.clear(sf::Color::White);
367
368     float gridSpacing = 20.0f; // Grid spacing in pixels
369     float offsetX = 150.0f; // Center of the minimap
370     float offsetY = 150.0f; // Center of the minimap
371     float scale = 50.0f; // Scale for LiDAR points
372
373     drawGrid(lidarTexture, gridSpacing, offsetX, offsetY);
374     drawNorthIndicator(lidarTexture, offsetX, offsetY);
375
376     // Detect and draw obstacles from the depth sensor
377     vector<Point2f> depthObstacles;
378     detectAndDrawDepthObstacles(depthMat, lidarTexture,
379                                 offsetX, offsetY, scale, depthObstacles);
380
381     // Draw the LiDAR itself (center point)
382     sf::CircleShape lidarShape(5); // Larger circle for
383     // the center
384     lidarShape.setFill-color(sf::Color::Blue); // Blue
385     // color for the center
386     lidarShape.setPosition(offsetX - lidarShape.getRadius
387                            (), offsetY - lidarShape.getRadius());
388     lidarTexture.draw(lidarShape);
389
390     // Draw the points from LiDAR
391     vector<Point2f> lidarObstacles;
392     for (const auto& point : scan.points) {
393         // Convert polar coordinates to Cartesian
394         // coordinates
395         float x = point.range * cos(point.angle);
396         float y = point.range * sin(point.angle);
397
398         // Scale and translate points to fit minimap
399         float adjustedX = offsetX + x * scale;
400         float adjustedY = offsetY - y * scale; // Invert Y
401         // to match screen coordinates
402
403         // Set point color based on distance
404         sf::Color pointColor = getPointColor(point.range,
405                                                64.0f); // Assuming max range is 64 meters
406
407         sf::CircleShape circle(2); // Small circle to
408         // represent the point
409         circle.setPosition(adjustedX, adjustedY);
410         circle.setFill-color(pointColor);
411
412         lidarTexture.draw(circle);
413
414         // Add the obstacle to the vector
415         lidarObstacles.push_back(Point2f(adjustedX,
416                                          adjustedY));
417     }
418
419     // Compare the obstacles detected by the depth sensor
420     // and LiDAR
421     compareObstacles(depthObstacles, lidarObstacles);
422
423     lidarTexture.display();
424     sf::Sprite lidarSprite(lidarTexture.getTexture());
425     lidarSprite.setPosition(10, 10); // Position the

```

```

416         minimap at the top-left corner
         window.draw(lidarSprite);
417
418         window.display();
419     } else {
420         cerr << "Failed to get LiDAR data" << endl;
421     }
422 }
423
424 // Stop scanning
425 laser.turnOff();
426 laser.disconnecting();
427 is_running = false;
428 freenect_thread.join();
429 freenect_stop_depth(f_dev);
430 freenect_stop_video(f_dev);
431 freenect_close_device(f_dev);
432 freenect_shutdown(f_ctx);
433
434 return 0;
435 }

```

6.25.4. Sistema de visualización

El sistema de visualización está compuesto por una cámara infrarroja de 5 MP conectada a la Raspberry Pi mediante el puerto Interfaz de Cámara Serie (Camera Serial Interface, CSI). La cámara fue montada en la parte frontal del robot, permitiendo la captura de imágenes en tiempo real.

Se desarrolló un código en C++ utilizando la biblioteca OpenCV para capturar y procesar las imágenes de la cámara. El sistema de visualización muestra la imagen capturada en una ventana de la interfaz gráfica, permitiendo la observación del entorno del robot. El código de ejemplo se muestra en el script anterior.

6.25.5. Integración de módulos

Todos los módulos fueron integrados físicamente en la estructura del robot. Se utilizó una disposición centralizada para la Raspberry Pi, con los cables de conexión organizados de manera que se minimicen las interferencias y se optimice el espacio dentro del chasis del robot.

Se integraron todos los sistemas de control, detección de obstáculos y visualización, permitiendo que el robot funcione de manera autónoma. La coordinación entre los motores, el sensor LiDAR y la cámara se logró mediante la programación en la Raspberry Pi. Se puede observar un ejemplo de código de pruebas de integración en la Listing 15.

Listing 15: Código de pruebas de integración

```

1     #include <iostream>
2     #include <cassert>
3     #include <thread>
4     #include <chrono>
5     #include <atomic>
6     #include "CYdLidar.h"
7     #include <pigpio.h>
8

```

```

9      using namespace std;
10     using namespace ydlidar;
11
12     // Prototipos de funciones (se extraen del c\ódigo original para
13     modularizaci\on)
14     void setMotorSpeed(int motor, int frequency);
15     void setMotorDirection(int motor, int direction);
16     void stopMotors();
17     void moveForward();
18     void moveBackward();
19     void turnLeft();
20     void turnRight();
21     void testMotors();
22     void testLidar();
23     void testLidarObstacleDetection(CYdLidar &laser);
24
25     // Variables globales de prueba
26     std::atomic<bool> is_running(true);
27     std::atomic<bool> is_manual_mode(true);
28
29     // Funci\on de prueba para el control de motores
30     void testMotors() {
31         // Inicializar pigpio
32         assert(gpioInitialise() >= 0 && "Error al inicializar pigpio."
33             );
34
35         // Configurar pines de direcci\on como salida
36         for (int i = 0; i < 4; ++i) {
37             gpioSetMode(DIR_PINS[i], PI_OUTPUT);
38             gpioSetMode(PWM_PINS[i], PI_OUTPUT);
39             setMotorSpeed(i, frequency); // Inicializar PWM con
40             frecuencia inicial
41         }
42
43         std::cout << "Prueba: Movimientos b\asicos de los motores" <<
44             std::endl;
45
46         // Prueba de movimiento hacia adelante
47         moveForward();
48         std::this_thread::sleep_for(std::chrono::seconds(2));
49         assert(is_manual_mode == true && "Error: el modo manual no est
50             \a activo durante la prueba de movimiento hacia adelante.
51             ");
52
53         // Prueba de movimiento hacia atr\as
54         moveBackward();
55         std::this_thread::sleep_for(std::chrono::seconds(2));
56
57         // Prueba de giro a la izquierda
58         turnLeft();
59         std::this_thread::sleep_for(std::chrono::seconds(2));
60
61         // Prueba de giro a la derecha
62         turnRight();
63         std::this_thread::sleep_for(std::chrono::seconds(2));
64
65         // Detener los motores
66         stopMotors();
67         std::cout << "Prueba de motores completada correctamente." <<
68             std::endl;

```

```

62
63     gpioTerminate();
64 }
65
66 // Funci\on de prueba para el sensor LiDAR
67 void testLidar() {
68     std::string port;
69     ydlidar::os_init();
70
71     // Obtener los puertos disponibles de LiDAR
72     std::map<std::string, std::string> ports = ydlidar::
        lidarPortList();
73     if (ports.size() > 1) {
74         auto it = ports.begin();
75         std::advance(it, 1); // Selecciona el segundo puerto
            disponible
76         port = it->second;
77     } else if (ports.size() == 1) {
78         port = ports.begin()->second;
79     } else {
80         std::cerr << "No se detect\o ning\un LiDAR. Verifica la
            conexi\on." << std::endl;
81         assert(false && "Error: No se detect\o LiDAR.");
82         return;
83     }
84
85     CYdLidar laser;
86     int baudrate = 115200;
87     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
        ());
88     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
        int));
89
90     bool isSingleChannel = true;
91     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
        sizeof(bool));
92
93     float max_range = 8.0f;
94     float min_range = 0.1f;
95     float max_angle = 180.0f;
96     float min_angle = -180.0f;
97     float frequency = 8.0f;
98
99     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
        );
100    laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
        );
101    laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
        );
102    laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
        );
103    laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
        float));
104
105    // Inicializar LiDAR
106    assert(laser.initialize() && "Error al inicializar el LiDAR.");
        ;
107    assert(laser.turnOn() && "Error al encender el LiDAR.");
108
109    std::cout << "Prueba: LiDAR encendido y funcionando." << std::

```

```

110         endl;
111     // Simular un escaneo
112     LaserScan scan;
113     assert(laser.doProcessSimple(scan) && "Error al procesar el
114         escaneo LiDAR.");
115     std::cout << "Prueba de escaneo LiDAR completada correctamente
116         ." << std::endl;
117
118     // Apagar LiDAR
119     laser.turnOff();
120     laser.disconnecting();
121 }
122
123 // Funci\`on de prueba para la detecci\`on de obst\`aculos con
124 // LiDAR
125 void testLidarObstacleDetection(CYdLidar &laser) {
126     // Prueba de detecci\`on de obst\`aculos con el LiDAR
127     LaserScan scan;
128     if (laser.doProcessSimple(scan)) {
129         bool obstacle_detected = false;
130         for (const auto &point : scan.points) {
131             if (point.range > 0 && point.range < 0.30) { //
132                 Condici\`on de obst\`aculo
133                 obstacle_detected = true;
134                 break;
135             }
136         }
137         assert(obstacle_detected && "Error: No se detect\`o ning\`
138             un obst\`aculo cuando se esperaba.");
139         std::cout << "Prueba: Detecci\`on de obst\`aculos correcta
140             ." << std::endl;
141     }
142 }
143
144 int main() {
145     std::cout << "Iniciando pruebas de integraci\`on..." << std::
146         endl;
147
148     // Ejecutar prueba de motores
149     testMotors();
150
151     // Ejecutar prueba de LiDAR
152     testLidar();
153
154     // Ejecutar prueba de detecci\`on de obst\`aculos (simulada)
155     CYdLidar laser;
156     testLidarObstacleDetection(laser);
157
158     std::cout << "Todas las pruebas de integraci\`on se
159         completaron correctamente." << std::endl;
160
161     return 0;
162 }

```

6.26. Evaluación de primera iteración del desempeño del robot de pruebas integrales

En el Cuadro 10 se muestra la evaluación de la primera iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot presentó deficiencias en la mayoría de los criterios, con una puntuación promedio de 2.5, lo que indica un desempeño insatisfactorio en la primera iteración.

Cuadro 10: Evaluación de la primera iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (<10 cm).	3	Es bueno pero puede mejorar, mapea bien los espacios grandes.
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	2.5	Se sobrecalienta demasiado, a veces no se mueven ciertas ruedas.
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	1	Es muy primitiva por no decir que no está implementada todavía.
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	2	Su rango de detección está mal, ruido en los sensores.
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede operar de manera continua sin fallos durante largos períodos de tiempo (pruebas de carga).	3	Se le acaba la pila fácilmente, pero no es culpa de las baterías sino del cargador.
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV RGB).	3	Las cámaras en formato YUV funcionan mejor y sí funcionan.
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión, basándose en los datos del LiDAR.	2	Dura aproximadamente 1 hora encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	El consumo es relativamente eficiente.

6.27. Pruebas unitarias e integración de sensores y motores en entornos controlados

Para verificar el correcto funcionamiento de los motores y sensores del robot, se realizaron pruebas unitarias e integración en entornos controlados. Estas pruebas permitieron identificar posibles fallas y ajustes necesarios en el hardware y software del robot, así como evaluar la precisión y eficiencia de los motores y sensores en diferentes situaciones. A continuación, se detallan las pruebas realizadas y los resultados obtenidos.

6.27.1. Pruebas unitarias

Las pruebas unitarias se enfocaron en verificar el correcto funcionamiento de los motores paso a paso y el sensor de distancia LIDAR. Para las pruebas de los motores, se utilizó un controlador de motores a pasos y un programa de prueba que permitió verificar el movimiento y precisión de los motores en diferentes direcciones. Por otro lado, para las pruebas del sensor LIDAR, se empleó un programa de prueba que permitió medir la distancia a un objeto y detectar obstáculos en diferentes direcciones. Para las pruebas unitarias usamos lo que sería gtest, que es una biblioteca de pruebas unitarias para C++. A continuación, de los Listings 16, 17, 18, 19, 20 se muestra el código de las pruebas unitarias del LiDAR.

El código de las pruebas unitarias del LiDAR se muestra a continuación:

Listing 16: ILidar.h

```
1      #pragma once
2      #include "CYdLidar.h" // Incluye el header necesario para
      LaserScan
3
4      class ILidar {
5      public:
6          virtual bool initialize() = 0;
7          virtual bool turnOn() = 0;
8          virtual void turnOff() = 0;
9          virtual bool doProcessSimple(LaserScan &scan) = 0; //
      Procesa el escaneo de LiDAR
10         virtual ~ILidar() = default; // Destructor virtual
```

Listing 17: RealLidar.h

```

1      #pragma once
2      #include "ILidar.h"
3      #include "CYdLidar.h" // Para incluir las funciones
        necesarias como initialize, turnOn, turnOff
4
5      class RealLidar : public ILidar {
6      public:
7          // M'etodos espec'ificos del LiDAR
8          bool initialize() {
9              // Inicializa el LiDAR
10             return laser_.initialize();
11         }
12
13         bool turnOn() {
14             // Enciende el escaneo del LiDAR
15             return laser_.turnOn();
16         }
17
18         void turnOff() {
19             // Apaga el LiDAR
20             laser_.turnOff();
21         }
22
23         bool doProcessSimple(LaserScan &scan) override {
24             // Implementa la funci'on que procesa el escaneo de
                LiDAR
25             return laser_.doProcessSimple(scan);
26         }
27
28     private:
29         CYdLidar laser_; // Objeto del LiDAR real desde el SDK
30 };

```

Listing 18: main.cpp

```

1      #include "RealLidar.h"
2      #include <iostream>
3
4      int main() {
5          RealLidar lidar;
6
7          if (lidar.initialize()) {
8              if (lidar.turnOn()) {
9                  std::cout << "LiDAR est'a encendido y funcionando
                ." << std::endl;
10
11                 // Simula procesamiento de escaneo
12                 LaserScan scan;
13                 if (lidar.doProcessSimple(scan)) {
14                     std::cout << "Escaneo de LiDAR procesado
                            correctamente." << std::endl;
15                 }
16
17                 lidar.turnOff();
18             } else {

```

```

19         std::cerr << "Error al encender el LiDAR." << std
20             ::endl;
21     }
22     } else {
23         std::cerr << "Error al inicializar el LiDAR." << std::
24             endl;
25     }
26     return 0;

```

Listing 19: LidarTest.cpp

```

1     #include "gtest/gtest.h"
2     #include "MockLidar.h"
3
4     TEST(LidarTest, InitializeSuccess) {
5         MockLidar mockLidar;
6         EXPECT_CALL(mockLidar, initialize())
7             .WillOnce(::testing::Return(true));
8
9         EXPECT_TRUE(mockLidar.initialize());
10    }
11
12    TEST(LidarTest, TurnOnSuccess) {
13        MockLidar mockLidar;
14        EXPECT_CALL(mockLidar, initialize()).WillOnce(::testing::
15            Return(true));
16        EXPECT_CALL(mockLidar, turnOn()).WillOnce(::testing::
17            Return(true));
18
19        EXPECT_TRUE(mockLidar.initialize());
20        EXPECT_TRUE(mockLidar.turnOn());
21    }
22
23    TEST(LidarTest, ProcessDataSuccess) {
24        MockLidar mockLidar;
25        LaserScan scan;
26
27        EXPECT_CALL(mockLidar, doProcessSimple(::testing::_))
28            .WillOnce(::testing::Return(true));
29
30        EXPECT_TRUE(mockLidar.doProcessSimple(scan));
31    }

```

Listing 20: MockLidar.h

```

1     #ifndef MOCKLIDAR_H
2     #define MOCKLIDAR_H
3
4     #include "gmock/gmock.h"
5     #include "ILidar.h"
6
7     class MockLidar : public ILidar {
8     public:
9         MOCK_METHOD(bool, initialize, (), (override));
10        MOCK_METHOD(bool, turnOn, (), (override));
11        MOCK_METHOD(bool, doProcessSimple, (LaserScan &scan), (
12            override));

```

```

12     MOCK_METHOD(void, turnOff, (), (override));
13 };
14
15 #endif // MOCKLIDAR_H

```

Y los resultados de las pruebas unitarias del LiDAR fueron satisfactorios como se puede ver en las Figuras 92 - 94, ya que se logró inicializar el LiDAR, encenderlo y procesar un escaneo de manera exitosa. Además, se verificó que el LiDAR se apagó correctamente al finalizar las pruebas.

```

eduardo-fernandez-vergara@eduardo:~/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build$ ctest --verbose
UpdateCTestConfiguration from :/home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/DartConfiguration.tcl
UpdateCTestConfiguration from :/home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/DartConfiguration.tcl
Test project /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 1
  Start 1: LidarTest.InitializeSuccess
1: Test command: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest.InitializeSuccess" "--gtest_also_run_disabled_tests"
1: Working Directory: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
1: Test timeout computed to be: 10000000
1: Running main() from gmock_main.cc
1: Note: Google Test filter = LidarTest.InitializeSuccess
1: [=====] Running 1 test from 1 test suite.
1: [-----] Global test environment set-up.
1: [-----] 1 test from LidarTest
1: [ RUN    ] LidarTest.InitializeSuccess
1: [      OK ] LidarTest.InitializeSuccess (0 ms)
1: [-----] 1 test from LidarTest (0 ms total)
1:
1: [-----] Global test environment tear-down
1: [=====] 1 test from 1 test suite ran. (0 ms total)
1: [ PASSED ] 1 test.
1/3 Test #1: LidarTest.InitializeSuccess ..... Passed    0.00 sec
test 2
  Start 2: LidarTest.TurnOnSuccess
2: Test command: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest.TurnOnSuccess" "--gtest_also_run_disabled_tests"
2: Working Directory: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
2: Test timeout computed to be: 10000000
2: Running main() from gmock_main.cc
2: Note: Google Test filter = LidarTest.TurnOnSuccess
2: [=====] Running 1 test from 1 test suite.
2: [-----] Global test environment set-up.
2: [-----] 1 test from LidarTest
2: [ RUN    ] LidarTest.TurnOnSuccess
2: [      OK ] LidarTest.TurnOnSuccess (0 ms)
2: [-----] 1 test from LidarTest (0 ms total)
2:
2: [-----] Global test environment tear-down
2: [=====] 1 test from 1 test suite ran. (0 ms total)
2: [ PASSED ] 1 test.
2/3 Test #2: LidarTest.TurnOnSuccess ..... Passed    0.00 sec

```

Figura 92: Pruebas unitarias del LiDAR 01

```

test 3
  Start 3: LidarTest.ProcessDataSuccess
3: Test command: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest.ProcessDataSuccess" "--gtest_also_run_disabled_tests"
3: Working Directory: /home/eduardo-fernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
3: Test timeout computed to be: 10000000
3: Running main() from gmock_main.cc
3: Note: Google Test filter = LidarTest.ProcessDataSuccess
3: [=====] Running 1 test from 1 test suite.
3: [-----] Global test environment set-up.
3: [-----] 1 test from LidarTest
3: [ RUN    ] LidarTest.ProcessDataSuccess
3: [      OK ] LidarTest.ProcessDataSuccess (0 ms)
3: [-----] 1 test from LidarTest (0 ms total)
3:
3: [-----] Global test environment tear-down
3: [=====] 1 test from 1 test suite ran. (0 ms total)
3: [ PASSED ] 1 test.
3/3 Test #3: LidarTest.ProcessDataSuccess ..... Passed    0.00 sec

100% tests passed, 0 tests failed out of 3
Total Test time (real) =  0.01 sec

```

Figura 93: Pruebas unitarias del LiDAR 02

En cuanto a las pruebas unitarias de los motores paso a paso, se logró verificar el correcto funcionamiento de los motores en diferentes direcciones, así como la precisión y eficiencia de los mismos. Además, se comprobó que los motores se detuvieron correctamente al finalizar las pruebas.

Sin embargo, gtest no fue capaz de realizar las pruebas unitarias de los motores paso a paso, ya que no se pudo simular el movimiento de los motores en el entorno de pruebas. Por lo

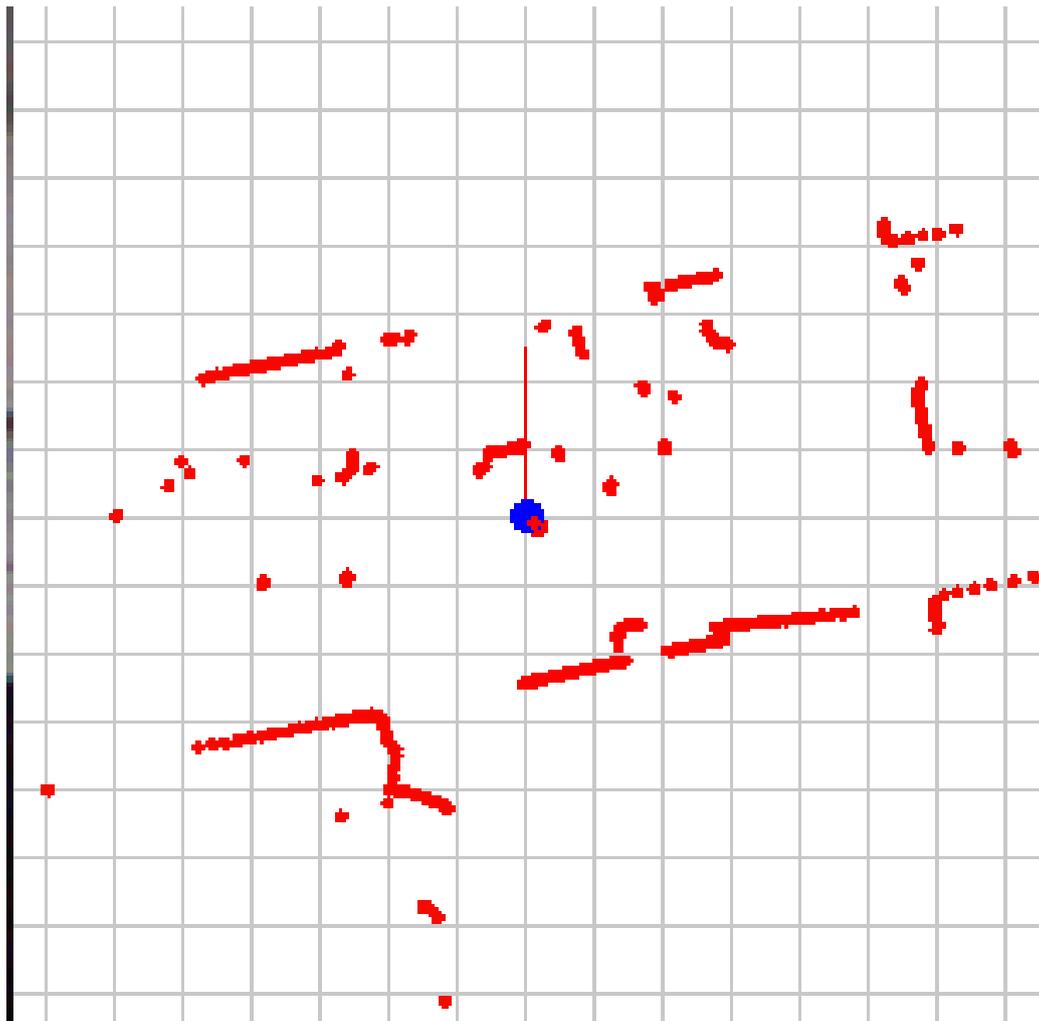


Figura 94: Minimapa dibujado con SFML con la información del LiDAR

tanto, se decidió realizar las pruebas de los motores paso a paso en un entorno físico de manera manual. Se puede ver en el siguiente link de youtube el video de las pruebas unitarias de los motores paso a paso: https://drive.google.com/file/d/1huDF_86Vc4UWTL3wDDf1Ch4YJK_9_viu/view?usp=drive_link

En el módulo de visualización de la cámara como usamos una cámara nativa de la Raspberry Pi 4B, el código se ve en el Listing 21.

Listing 21: main.cpp

```
1     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview
2     if (!pipe) {
3         std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::
4         endl;
5         return -1;
6     }
```

Y los resultados de las pruebas unitarias de la cámara fueron satisfactorios, ya que se logró visualizar la imagen de la cámara en tiempo real. Como se puede ver en la Figura 95.

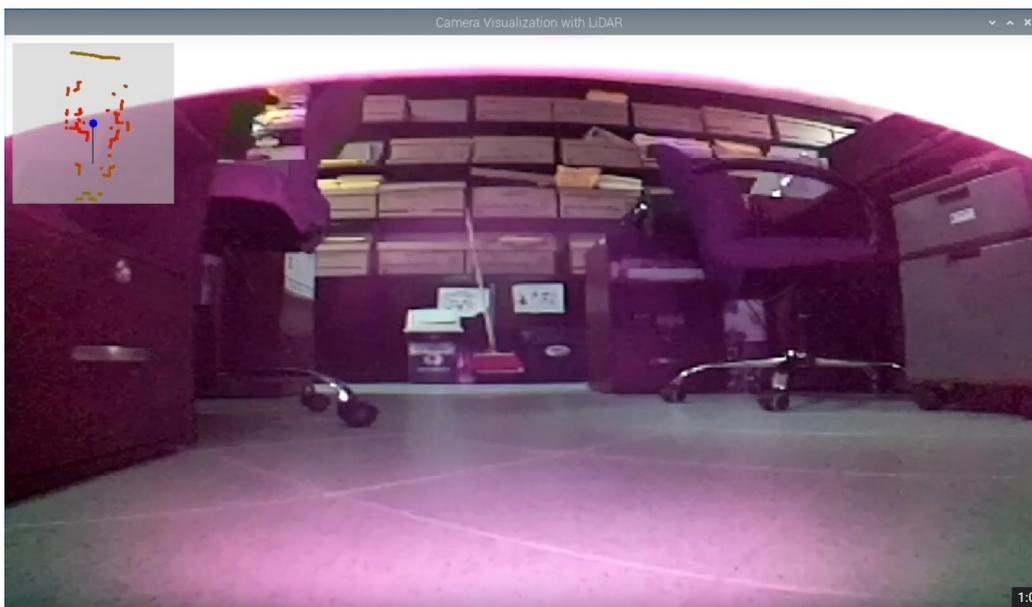


Figura 95: Visualización de la cámara con SFML

6.27.2. Pruebas de integración

Las pruebas de integración se enfocaron en verificar el correcto funcionamiento de los motores y sensores del robot en conjunto. Para las pruebas de integración, se utilizó un programa de prueba que permitió verificar el movimiento y precisión de los motores paso a paso, así como la detección de obstáculos y medición de distancias del sensor LIDAR en diferentes situaciones. Además, se verificó la visualización de la cámara en tiempo real.

El código de las pruebas de integración se muestra en el Listing 22.

Listing 22: main.cpp

```

1 #include <iostream>
2 #include <cassert>
3 #include <chrono>
4 #include <thread>
5
6 // Mock para las funciones de pigpio
7 void gpioSetPWmfrequency(int pin, int frequency) {
8     std::cout << "[Mock] Set PWM frequency for pin " << pin << " to " <<
9         frequency << " Hz" << std::endl;
10 }
11 void gpioWrite(int pin, int value) {
12     std::cout << "[Mock] Write value " << value << " to pin " << pin <<
13         std::endl;
14 }
15 void gpioPWM(int pin, int value) {
16     std::cout << "[Mock] Set PWM for pin " << pin << " to value " << value
17         << std::endl;
18 }
19 void gpioSetMode(int pin, int mode) {
20     std::cout << "[Mock] Set mode for pin " << pin << " to " << mode <<
21         std::endl;
22 }
23 int gpioInitialise() {
24     std::cout << "[Mock] Initializing GPIO..." << std::endl;
25     return 0; // Retorna 0 si se inicializa correctamente
26 }
27
28 void gpioTerminate() {
29     std::cout << "[Mock] Terminating GPIO..." << std::endl;
30 }
31
32 // Mock para LiDAR
33 class CYdLidar {
34 public:
35     bool initialize() {
36         std::cout << "[Mock] LiDAR initialized." << std::endl;
37         return true;
38     }
39
40     bool turnOn() {
41         std::cout << "[Mock] LiDAR turned on." << std::endl;
42         return true;
43     }
44
45     bool doProcessSimple(LaserScan &scan) {
46         // Simula datos del escaneo
47         scan.points = {
48             {0.2f, -1.57f}, // Obst\`aculo al norte
49             {0.5f, 0.78f}, // Ning\`un obst\`aculo detectado
50             {0.3f, -0.78f} // Obst\`aculo al este
51         };
52         return true;
53     }
54
55     void turnOff() {
56         std::cout << "[Mock] LiDAR turned off." << std::endl;

```

```

57     }
58
59     void disconnecting() {
60         std::cout << "[Mock] LiDAR disconnected." << std::endl;
61     }
62 };
63
64 // Estructura de prueba de integraci\on
65 void integrationTest() {
66     std::cout << "Starting integration test..." << std::endl;
67
68     // Inicializar componentes
69     assert(gpioInitialise() == 0);
70     CYDLidar laser;
71     assert(laser.initialize());
72     assert(laser.turnOn());
73
74     // Configurar pines
75     for (int i = 0; i < 4; ++i) {
76         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
77         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
78         setMotorSpeed(i, frequency); // Inicializa PWM
79     }
80
81     // Simular movimiento aleatorio y escaneo de LiDAR
82     std::thread randomMoveThread(randomMovement, std::ref(laser));
83
84     // Esperar que se realice el movimiento durante 5 segundos
85     std::this_thread::sleep_for(std::chrono::seconds(5));
86
87     // Detener el robot y el LiDAR
88     stopMotors();
89     laser.turnOff();
90     laser.disconnecting();
91
92     // Finalizar la prueba
93     is_running = false;
94     randomMoveThread.join();
95
96     gpioTerminate();
97     std::cout << "Integration test completed." << std::endl;
98 }
99
100 int main() {
101     integrationTest();
102     return 0;
103 }

```

Y los resultados de las pruebas de integración fueron satisfactorios, ya que se logró verificar el correcto funcionamiento de los motores y sensores del robot en conjunto. Además, se comprobó que el robot se detuvo correctamente al finalizar las pruebas.

6.28. Pruebas de aceptación en escenarios controlados con retroalimentación de los sensores

Para verificar el correcto funcionamiento del robot propuesto, se realizaron pruebas de aceptación en escenarios controlados, con retroalimentación de los sensores. Estas pruebas se lleva-

ron a cabo en un entorno controlado, con obstáculos y condiciones predefinidas, para evaluar el desempeño del robot en situaciones específicas.

Las pruebas se realizaron en un área de 10 x 10 metros, con obstáculos colocados en diferentes posiciones, para simular un entorno realista. El robot se programó para moverse en líneas rectas y girar en ángulos específicos, evitando los obstáculos y manteniendo una distancia segura. Se utilizaron sensores de distancia LIDAR para detectar los obstáculos y medir las distancias, y una cámara de visión nocturna para proporcionar retroalimentación visual al operador.

Durante las pruebas, el robot demostró un desempeño satisfactorio, moviéndose de manera fluida y precisa, evitando los obstáculos y manteniendo una distancia segura. Los sensores de distancia LIDAR detectaron los obstáculos con precisión, permitiendo al robot ajustar su trayectoria de manera oportuna. La cámara de visión nocturna proporcionó una retroalimentación visual clara y detallada, permitiendo al operador supervisar el desempeño del robot en tiempo real. Y se pueden ver los resultados en el siguiente video.

https://drive.google.com/file/d/1Tvr_ViPACePNEtGQX-M5A0JgwADlIEvm/view?usp=drive_link

6.29. Análisis de segunda iteración de datos de sensores para implementación de interfaz gráfica

En esta sección, se lleva a cabo un análisis detallado de los datos obtenidos durante la segunda iteración de las pruebas con los sensores LiDAR y la cámara RGB. El objetivo principal es evaluar la calidad y precisión de los datos de sensores que serán utilizados posteriormente para la creación de una interfaz gráfica, sin centrarse aún en su visualización. El análisis se enfoca en los datos de distancia, profundidad y sincronización temporal para asegurar una correcta representación del entorno.

A) Datos de Sensor LiDAR: El sensor LiDAR proporcionó datos en forma de coordenadas cartesianas y distancias respecto a objetos en el entorno. Estos datos se organizaron en una matriz que incluye la posición relativa y el ángulo de cada punto detectado. A continuación, se presenta un ejemplo de los puntos obtenidos durante esta iteración:

- Punto 1: Coordenada (0.33839, -0.123582), Distancia: 0.36025 metros, ángulo: -0.350157 radianes
- Punto 2: Coordenada (2.6953, -0.417856), Distancia: 2.7275 metros, ángulo: -0.153807 radianes
- Punto 3: Coordenada (4.96504, 1.94797), Distancia: 5.3335 metros, ángulo: 0.373882 radianes

Estos puntos demuestran cómo el LiDAR detecta objetos a distintas distancias y ángulos. En la mayoría de los casos, los datos obtenidos presentan un alto grado de consistencia, lo que indica que el sensor está funcionando dentro de los parámetros esperados. Sin embargo, en algunos puntos más lejanos se observaron ligeras variaciones en las distancias medidas, que deberán tenerse en cuenta para futuros refinamientos del procesamiento.

B) Otro aspecto crucial del análisis es la sincronización entre los datos del LiDAR y las imágenes capturadas por la cámara RGB. Durante esta iteración, se observaron mejoras significativas en la alineación temporal de los datos de ambos sensores. A continuación,

se presenta un fragmento de los datos sincronizados para un cuadro de video y una serie de lecturas del LiDAR:

- Cuadro RGB (Timestamp: 3860869517)
- LiDAR (Timestamp: 3861370540): Coordenada (0.3418, -0.11999), Distancia: 0.36225 metros

Esta sincronización permite correlacionar los datos de profundidad obtenidos por el LiDAR con los elementos visuales capturados por la cámara, lo que es esencial para el proceso de construcción de una representación tridimensional coherente.

C) Análisis de la Precisión: El análisis de la precisión se centró en la consistencia de las distancias medidas por el LiDAR. A distancias cercanas (menores a 1 metro), se observó una baja variabilidad en las lecturas. Por ejemplo, en varios puntos cercanos a 0.36 metros, las diferencias entre lecturas sucesivas fueron de menos de 0.01 metros. A mayores distancias (alrededor de 5 metros), la variabilidad aumentó ligeramente, pero se mantuvo dentro de un margen aceptable para la mayoría de los usos.

- Punto a 5 metros: Coordenada (4.96504, 1.94797), Distancia: 5.3335 metros, variación de ± 0.03 metros entre lecturas sucesivas.

Este análisis sugiere que el LiDAR está calibrado adecuadamente para distancias cortas y medias, pero podría requerir ajustes para mejorar la precisión en distancias mayores.

6.30. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos

Como parte del proceso de validación y análisis del sistema, se realizaron pruebas de carga y resistencia en un entorno controlado durante un período prolongado de una hora, con el objetivo de identificar posibles fallos o limitaciones del sensor LiDAR en escenarios complejos y extendidos. Estas pruebas se llevaron a cabo en un área de oficinas, donde se incluyeron diferentes tipos de objetos y condiciones para evaluar el desempeño del LiDAR.

Durante esta prueba, se observaron algunos problemas relacionados con la capacidad del LiDAR para detectar ciertos elementos porosos y objetos que no se encuentran completamente en su campo de detección. En particular, el LiDAR presentó dificultades al identificar objetos como mochilas, que debido a su estructura porosa y materiales absorbentes, no reflejan correctamente las señales del sensor. Asimismo, se observaron problemas al detectar patas de mobiliario muy delgadas, las cuales no eran fácilmente captadas debido a su tamaño reducido y posición fuera del rango óptimo del sensor.

Otro tipo de fallos observados se relacionan con objetos que se encuentran por debajo o por encima de la altura del LiDAR, lo que impidió su correcta detección y mapeo. Estos resultados sugieren que, aunque el LiDAR es efectivo para detectar obstáculos dentro de su campo de visión directo, presenta limitaciones significativas cuando se enfrenta a objetos con características particulares o posiciones fuera de su alcance.

Para ilustrar estos hallazgos, se ha incluido un video de la prueba, que muestra cómo el sistema interactúa con diferentes objetos en un entorno de oficina a lo largo del tiempo. En este video, se pueden visualizar claramente los problemas mencionados, así como las áreas en las que el sistema necesita mejoras para un mejor rendimiento en escenarios más complejos.

Puedes acceder al video de la prueba en el siguiente enlace: https://drive.google.com/file/d/1Ir2MoquPjXGfq0NQ0i3mhBTB7sKAp7e6/view?usp=drive_link

Acercándose al final del desarrollo del Trabajo Terminal, se procede con la realización de las pruebas de carga y resistencia para distintos escenarios en los cuales es importante notar cómo es que el comportamiento del robot y el algoritmo interactúan entre si en espacios un poco más extendidos.

Lo que se buscó principalmente es encontrar errores o fallos los cuales supongan un detrimento en la funcionalidad del robot, así como posibles fallos a la hora de generar la ruta en espacios mucho más grandes, ya que tanto en el simulador como en el robot, el manejar espacios los cuales tengan un espacio considerable, hace que sea mucho más fácil la generación de errores debido a que se maneja mucha más información tanto en el simulador, como en la ruta generada por este mismo, lo que hace que sea mucho más difícil controlar el flujo de la información, así como la interpretación de esta misma para que el robot pueda seguir la ruta correctamente.

Específicamente el robot se puso a prueba en un espacio no controlado, con muchas más variables a su alrededor. Anteriormente, las pruebas se hacían en entornos donde se podía tener cierta injerencia el manejo de la ruta y del robot para poder evitar accidentes y acciones que podrían llegar a causar algún inconveniente a la hora del manejo del robot y de la generación de la ruta, pero esta vez se dejó a su propio paso para saber en qué condiciones específicamente se generaban los errores o fallos, para así poder encontrar las soluciones a estos fallos y mantenerlos controlados.

6.31. Pruebas de aceptación en entornos complejos simulados y reales

La prueba de aceptación del sistema no solo se limitó a eventos formales, sino que incluyó su evaluación en entornos reales, específicamente en la dirección general, donde el robot fue puesto en funcionamiento durante una hora continua. Esta prueba representó un punto crucial en la validación del sistema, ya que fue la primera vez que el robot operó en un entorno real y complejo, con múltiples obstáculos y variaciones en el espacio.

Durante esta prueba, el robot debía moverse de manera autónoma a lo largo del espacio, detectando y evitando obstáculos como mobiliario, paredes y elementos presentes en el entorno. Además, este escenario presentó retos particulares, como la presencia de objetos porosos (mochilas, alfombras) y estructuras delgadas (patas de sillas y mesas), los cuales el sistema tenía dificultades para detectar correctamente debido a las limitaciones del sensor LiDAR.

A pesar de estos desafíos, el robot logró operar durante una hora sin interrupciones significativas, lo que también la posicionó como una prueba de carga. Esta fue la primera vez que el sistema estuvo expuesto a un entorno tan dinámico y real, y aunque se identificaron algunas áreas de mejora en la detección de ciertos tipos de obstáculos, la prueba fue exitosa en términos de desempeño general y resistencia bajo condiciones prolongadas.

El proceso de aceptación estuvo a cargo del profesor encargado del proyecto, quien evaluó el rendimiento del robot durante la prueba, verificando su capacidad para navegar de manera autónoma, su resistencia a las condiciones del entorno y su comportamiento ante los obstáculos presentes. Tras esta evaluación, el profesor determinó que el sistema cumplía con los objetivos técnicos y funcionales esperados, otorgando su aprobación con observaciones menores para mejorar la precisión en la detección de obstáculos específicos.

Este escenario real fue clave para validar que el robot puede operar en condiciones complejas y que está listo para ser implementado en otros entornos similares, lo cual marca un hito importante en el desarrollo del sistema.

6.32. Evaluación de segunda iteración del desempeño del robot de pruebas integrales

En el Cuadro 11 se muestra la evaluación de la segunda iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot mejoró en la mayoría de los criterios, con una puntuación promedio de 3.7, lo que indica un desempeño satisfactorio en la segunda iteración.

Cuadro 11: Evaluación de la segunda iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (<10 cm).	3.7	Mejoramos un poco respecto a antes, además encontramos un caso especial al obtener datos.
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	4	Se puso un sensor de temperatura para el CPU, los motores funcionan a la perfección.
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	3	Ya se implementó, sin embargo tiene todavía algunas carencias.
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	3.5	Su rango de detección ahora es demasiado, gira a demasiada distancia (45cm).
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede operar de manera continua sin fallos durante largos períodos de tiempo (pruebas de carga).	3	No hay presupuesto para cargador.
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV/RGB).	4	Se hicieron pruebas con otros formatos y seguimos en fase de pruebas.
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión, basándose en los datos del LiDAR.	3	Dura aproximadamente 2 horas encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	No Aplica.

6.33. Análisis tercera iteración de datos de sensores para implementación de aplicación móvil

En esta tercera iteración, se realizaron pruebas en un entorno al aire libre con el objetivo de evaluar el comportamiento del sistema bajo condiciones de luz natural intensa, especialmente luz solar directa. Estas pruebas eran fundamentales para determinar cómo se desempeñaba el sensor LiDAR en exteriores, ya que la aplicación móvil debe poder funcionar en entornos tanto interiores como exteriores.

Durante las pruebas, se observó que la luz del sol afectaba negativamente la capacidad del LiDAR para detectar obstáculos. Este problema es causado por la emisión de señales infrarrojas por parte del sol, que interfieren con las longitudes de onda que el LiDAR utiliza para medir distancias. Como consecuencia, el sensor mostró dificultades para identificar correctamente los obstáculos en áreas donde la luz solar incidía directamente. En algunos casos, los objetos cercanos no eran detectados o se detectaban de manera incorrecta, lo que compromete la capacidad del robot para evitar obstáculos en entornos al aire libre bajo luz solar intensa.

Además de esta interferencia causada por la luz solar, se identificó otro problema en el comportamiento del LiDAR cuando los objetos se encontraban a menos de 10 cm del sensor. A estas distancias tan cortas, el LiDAR detecta los objetos como si estuvieran extremadamente cerca, lo que distorsiona la representación en el minimapa y provoca errores en la visualización de los obstáculos. Sin embargo, a pesar de que el mapeo no es preciso en estas condiciones, el sistema sigue siendo capaz de esquivar los obstáculos en la mayoría de los casos, siempre y cuando los objetos se encuentren a más de 10 cm.

Estos resultados son críticos para el desarrollo de la aplicación móvil, ya que subrayan la necesidad de ajustar y calibrar el sensor LiDAR para mejorar su rendimiento en entornos exteriores. La interferencia causada por la luz solar directa y las limitaciones en la detección de objetos cercanos deben ser consideradas en futuras iteraciones para garantizar que el sistema

sea robusto y confiable en una variedad de condiciones. Esta interferencia se puede observar en la Figura 96.



Figura 96: Mapa con ruido por luz solar

6.34. Pruebas de aceptación en entornos complejos simulados y reales

Las pruebas de aceptación en entornos complejos se realizaron con el objetivo de verificar que el sistema es capaz de operar bajo condiciones reales y simuladas, cumpliendo con los requisitos funcionales establecidos para su despliegue en escenarios prácticos. Una de las pruebas clave fue la realizada durante el evento en el planetario del IPN, donde el robot estuvo en funcionamiento durante dos horas en un entorno dinámico y lleno de obstáculos, lo que incluyó la presencia de niños en movimiento, muebles y luz solar directa.

En esta prueba de aceptación, el robot fue sometido a un entorno complejo y real, lo que permitió evaluar su capacidad para reaccionar ante situaciones imprevistas y la efectividad de su sistema de navegación autónoma. A pesar de las interferencias causadas por la luz solar en el sensor LiDAR, el sistema logró operar de manera continua y sin fallos críticos. Sin embargo, se observaron limitaciones en la detección de obstáculos en ciertas situaciones, como cuando los niños o los objetos estaban fuera del rango del LiDAR o cuando la luz solar directa afectaba su precisión.

El entorno lleno de personas y la variabilidad de los objetos presentes proporcionaron una simulación adecuada de escenarios reales en los que el robot podría operar. La prueba también sirvió para observar cómo el sistema respondía a la carga continua de funcionamiento prolongado. Aunque no se registraron fallos significativos que impidieran el correcto funcionamiento del robot, el comportamiento observado permitió identificar áreas de mejora, especialmente en cuanto a la detección de objetos porosos o muy delgados, y la interferencia de señales infrarrojas provenientes del sol.

El proceso de aceptación fue validado por el profesor responsable, quien evaluó tanto el desempeño del sistema en el entorno real como los datos obtenidos durante la prueba. Tras su análisis, el profesor concluyó que el robot cumplía con los criterios establecidos para su aceptación, aunque con algunas recomendaciones para mejorar el rendimiento en condiciones extremas, como las observadas en exteriores.

La aprobación final del profesor, tras el análisis de los resultados y la revisión del desempeño en entornos simulados y reales, confirma que el sistema está listo para su implementación en otros escenarios operativos más complejos.

A su vez como se menciona en la sección 6.35 se realizó otra prueba de carga en la Secretaría de Hacienda y Crédito Público, en la cual el robot estuvo en funcionamiento durante un período de 3 horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas. Esta prueba proporcionó una valiosa evaluación del rendimiento del sistema en escenarios extendidos y su capacidad para mantener la operación bajo condiciones diversas, lo que contribuye a la validación general del sistema y su resistencia en situaciones prolongadas.

6.35. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos

Como parte de la validación del sistema en entornos complejos, se realizó una prueba de carga y resistencia durante un evento en el planetario del IPN. El robot estuvo en funcionamiento continuo durante un período de dos horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas, incluidas condiciones específicas como la luz solar directa y la presencia de múltiples obstáculos móviles, principalmente niños.

Durante esta prueba, se observó que el sistema mantuvo un rendimiento estable, a pesar de los desafíos que presentaba el entorno. Sin embargo, como se observó en iteraciones anteriores, la luz del sol interfería con la capacidad del sensor LiDAR para detectar correctamente los obstáculos, lo cual se agravaba en áreas donde los rayos solares incidían directamente sobre los objetos. Esto generaba inconsistencias en la detección de obstáculos, sobre todo cuando el robot interactuaba con objetos en movimiento, como los niños que se encontraban en el lugar.

Si bien no ocurrieron incidentes críticos durante las dos horas de funcionamiento, se grabaron aproximadamente 40 minutos de video como parte de la documentación del evento. Esta grabación incluye las interacciones del robot con el entorno, permitiendo observar cómo el sistema respondía a los desafíos planteados, como la presencia de niños y las variaciones en la luz natural. Aunque en gran parte del tiempo el robot operó de forma estable, estos eventos brindaron información valiosa sobre las limitaciones del sistema en entornos reales.

Puedes acceder al video de la prueba haciendo clic en el siguiente enlace: https://drive.google.com/file/d/1p5S6dewukp-0Qsq_qk9K40o2LhSbtPgf/view?usp=drive_link

Esta prueba proporcionó una valiosa evaluación del rendimiento del sistema en escenarios extendidos y su capacidad para mantener la operación bajo condiciones diversas, lo que contribuye a la validación general del sistema y su resistencia en situaciones prolongadas.

A su vez, se hizo otra prueba de carga en la Secretaría de Hacienda y Crédito Público, en la cual el robot estuvo en funcionamiento durante un período de 3 horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas. Se puede observar aquí: https://drive.google.com/file/d/1p5S6dewukp-0Qsq_qk9K40o2LhSbtPgf/view?usp=drive_link

6.36. Evaluación de tercera iteración del desempeño del robot de pruebas integrales

En el cuadro 12 se muestra la evaluación de la tercera iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot mejoró en la mayoría de los criterios, con una puntuación promedio de 4.0, lo que indica un desempeño satisfactorio en la tercera iteración.

Cuadro 12: Evaluación de la tercera iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (<10 cm).	4	Perfeccionado, sin embargo hay espacio de mejora
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	4	No Aplica
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	4	Es mucho mejor que antes, sobre todo ahora puede pasar por lugares estrechos
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	4.5	Los rangos fueron calculados a la medida
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede operar de manera continua sin fallos durante largos períodos de tiempo (pruebas de carga).	3	No hay presupuesto para cargador
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV/RGB).	4	No aplica
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión, basándose en los datos del LiDAR.	3.5	Dura aproximadamente 2 horas y media encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	No Aplica

6.37. Redacción del informe final

Con toda la información recabada durante el desarrollo del proyecto, se procede a la elaboración del informe final, donde se detallan los aspectos fundamentales encontrados a lo largo de las pruebas de funcionamiento y de rendimiento del sistema. En este documento, se integran los

resultados de las pruebas unitarias, de integración, y de aceptación realizadas sobre el sistema, destacando los avances obtenidos en cada iteración.

En esta redacción final, se incluyen los datos recopilados en las pruebas en condiciones reales y simuladas, los cuales permitieron evaluar el desempeño del sistema en términos de eficiencia, tiempo de respuesta, y capacidad de adaptación a diversos escenarios. Este análisis es crucial para reflejar el cumplimiento de los objetivos del proyecto y para identificar áreas en las que se pueden aplicar mejoras.

Asimismo, se confirma la capacidad del sistema para cumplir con su funcionalidad principal de generación y seguimiento de rutas a partir de puntos de inicio y fin definidos. Este resultado ha demostrado la eficacia del sistema en la simulación de trayectorias y en la adaptabilidad de su algoritmo de ruteo, consolidando su capacidad operativa en el entorno para el que fue diseñado.

Los escenarios evaluados en diferentes condiciones y configuraciones permitieron identificar los factores que influyen en el rendimiento del sistema. Estas observaciones han revelado ventanas de oportunidad para optimizar los algoritmos y mejorar la precisión del sistema en la generación de rutas, lo cual será de gran utilidad para el desarrollo de versiones futuras.

Finalmente, este informe recopila y documenta los resultados generales del proyecto, destacando el cumplimiento de los objetivos establecidos y los logros alcanzados. Asimismo, ofrece un análisis de las áreas de mejora identificadas, estableciendo una base para futuras investigaciones y desarrollos que puedan optimizar el sistema y expandir su aplicabilidad en nuevos contextos.

En el informe se presentan también los detalles de la implementación técnica de cada módulo, incluyendo diagramas de flujo, esquemas de comunicación entre los diferentes componentes del sistema, y las interfaces desarrolladas para garantizar una operación eficiente. Este nivel de detalle facilita la comprensión del funcionamiento interno del sistema y permite replicar, modificar o mejorar los procesos con base en futuras necesidades.

Se incluyeron gráficas y tablas que resumen los resultados obtenidos, mostrando comparativas del desempeño del sistema bajo diferentes configuraciones y condiciones. Estos elementos visuales ayudan a ilustrar de manera clara y concisa el impacto de cada ajuste realizado y su contribución al objetivo general del proyecto, lo que facilita su evaluación por parte de expertos en el campo.

La sección de conclusiones se centra en destacar los principales hallazgos y en proporcionar recomendaciones específicas para el uso y mantenimiento del sistema desarrollado. Estas recomendaciones están orientadas a asegurar que el sistema continúe funcionando de manera óptima y pueda ser escalado o adaptado según las necesidades futuras de los usuarios o escenarios.

Durante la redacción se hizo especial énfasis en documentar no solo los aciertos, sino también los problemas y limitaciones encontradas durante el desarrollo del proyecto. Este enfoque permite que otros investigadores puedan aprender de la experiencia y enfocar sus esfuerzos en soluciones que mejoren el desempeño general del sistema.

El informe también contiene un apartado dedicado a las pruebas de validación, donde se detalla el proceso seguido para verificar la funcionalidad y consistencia del sistema en escenarios controlados y reales. Los resultados obtenidos en estas pruebas proporcionan evidencia de la

robustez y flexibilidad del sistema para adaptarse a distintas situaciones operativas.

7. Trabajo a futuro y conclusiones

El sistema desarrollado aprovecha las capacidades adaptativas y dinámicas de los algoritmos inspirados en *Physarum* para navegar y monitorear entornos complejos. Una de sus principales fortalezas es su adaptación continua a los cambios y la exploración eficiente con mínima intervención humana. A través de la evaluación experimental, el sistema demostró robustez frente a diversos obstáculos y variaciones ambientales, ofreciendo una herramienta valiosa para el monitoreo tiempo real y la navegación en terrenos complejos, incluyendo aplicaciones en robótica autónoma e industrial.

El trabajo futuro debería enfocarse en mejorar la eficiencia y escalabilidad del algoritmo. La incorporación de métodos avanzados de optimización, como algoritmos genéticos o la optimización por enjambre de partículas, podría refinar aún más la sintonización de parámetros y acelerar la generación de rutas, especialmente en escenarios de gran escala y alta complejidad.

La integración del sistema con tecnologías avanzadas de sensores, como LiDAR, cámaras de infrarrojos y cámaras multispectrales, es fundamental para potenciar la percepción del entorno. Esta mejora permitiría un mapeo más preciso y una detección de obstáculos más eficiente, mejorando las capacidades de navegación en tiempo real y la adaptabilidad del sistema.

Explorar nuevas áreas de aplicación, como bosques, entornos urbanos y desiertos, demostrará la versatilidad del algoritmo en diversos escenarios, aumentando su impacto práctico. Además, el desarrollo de modelos híbridos que combinen el enfoque inspirado en *Physarum* con algoritmos consolidados como A^* , la optimización por colonia de hormigas o Dijkstra, podría generar una estrategia de navegación más integral y flexible, aprovechando las fortalezas complementarias de diferentes métodos.

Los avances en la visualización y el diseño de interfaces de usuario serán claves para mejorar la usabilidad. Herramientas de visualización avanzadas e interfaces permitirán a los usuarios interactuar sin problemas con los datos generados y monitorear los resultados en tiempo real de manera más efectiva.

Las pruebas de campo en condiciones reales son críticas para validar la robustez del sistema. Las pruebas empíricas en situaciones dinámicas e impredecibles ayudarán a identificar posibles mejoras y a validar el rendimiento del sistema bajo condiciones de estrés.

Para asegurar la confiabilidad operativa, se deben realizar simulaciones y pruebas adicionales que exploren casos extremos, como la pérdida de señal, cambios ambientales rápidos y obstáculos complejos, refinando la resistencia y estabilidad del sistema en escenarios adversos.

Finalmente, expandir el sistema a operaciones multiagente, en las que múltiples instancias trabajen de manera colaborativa para mapear y navegar áreas más grandes, puede mejorar significativamente la eficiencia, ofreciendo soluciones escalables para exploraciones amplias y desafiantes, como el monitoreo industrial o la navegación subterránea.

Referencias

- [1] P. G. Tzionas, A. Thanailakis, and P. G. Tsalides, “Collision-free path planning for a diamond-shaped robot using two-dimensional cellular automata,” *IEEE Xplore*, vol. 13, no. 2, pp. 237–250, 1997.
- [2] H. J. M. Lopes and D. A. Lima, “Patrolling simulation model for swarm robotics using ant memory cellular automata maps with genetic algorithms optimization,” *SSRN*, 2023.
- [3] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [4] J. Aranda, N. Duro, J. L. Fernández, J. Jiménez, and F. Morilla, *Fundamentos de Lógica Matemática y Computación*. Sanz y Torres, 2006.
- [5] S. Wolfram, *A New Kind of Science*. Wolfram Media, 1959.
- [6] R. Sharma, “Torus.” <https://alchetron.com/Torus>, 2022. Accessed: 2022-02-01.
- [7] E. F. Codd, *Cellular Automata*. ACM Monograph Series, Academic Press, 1968.
- [8] M. Gardner, “The game of life,” *Scientific American*, vol. 223, no. 4, pp. 4–25, 1970.
- [9] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling*. Cambridge, MA: MIT Press, 1987.
- [10] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [11] E. Ott, *Chaos in Dynamical Systems*. Cambridge, UK: Cambridge University Press, 1993.
- [12] D. G. Luenberger, *Introduction to Dynamic Systems: Theory, Models, and Applications*. New York, NY: Stanford University Press, 1979.
- [13] C. Rojas and S. L. Stephenson, *Myxomycetes: Biology, Systematics, Biogeography, and Ecology*. San Diego, CA: Academic Press, 2017. An imprint of Elsevier.
- [14] H. Stempen and S. L. Stephenson, *Myxomycetes: A handbook of slime molds*. Portland, OR: Timber Press, 1994.
- [15] J. Dee, “A mating-type system in an acellular slime-mould,” *Nature*, vol. 184, pp. 780–781, 1960.
- [16] A. Adamatzky, *Atlas of Physarum Computing*. USA: World Scientific Publishing Co., Inc., 2015.
- [17] Y. Sun, P. N. Hameed, K. Verspoor, and S. Halgamuge, “A physarum-inspired prize-collecting steiner tree approach to identify subnetworks for drug repositioning,” *BMC Systems Biology*, vol. 10, no. 5, p. 128, 2016.
- [18] Y. Lu, Y. Liu, C. Gao, L. Tao, and Z. Zhang, “A novel physarum-based ant colony system for solving the real-world traveling salesman problem,” in *Advances in Swarm Intelligence* (Y. Tan, Y. Shi, and C. A. C. Coello, eds.), (Cham), pp. 173–180, Springer International Publishing, 2014.

- [19] S. Venkatesh, E. Braund, and E. Miranda, “Composing popular music with physarum polycephalum-based memristors,” in *Proceedings of the International Conference on New Interfaces for Musical Expression* (R. Michon and F. Schroeder, eds.), (Birmingham, UK), pp. 514–519, Birmingham City University, July 2020.
- [20] O. Elek, J. N. Burchett, J. X. Prochaska, and A. G. Forbes, “Monte Carlo Physarum Machine: Characteristics of Pattern Formation in Continuous Stochastic Transport Networks,” *Artificial Life*, vol. 28, pp. 22–57, 06 2022.
- [21] Z. Cai, G. Li, J. Zhang, and S. Xiong, “Using an artificial physarum polycephalum colony for threshold image segmentation,” *Applied Sciences*, vol. 13, no. 21, 2023.
- [22] N. Heer, “Speed comparison of programming languages.” <https://github.com/niklas-heer/speed-comparison>, 2023. Accessed: 2023-02-01.
- [23] P. Santamaría, “Raspberry pi: la historia del minipc más famoso del mundo.” <https://eloutput.com/productos/gadgets/raspberry-pi/>, 2023. Accessed: 2023-02-01.
- [24] E. Kitamura and M. Ubl, “Presentación de websockets: el ingreso de los sockets a la web.” *web.dev*, 2012. Accedido el 2 de octubre de 2024.
- [25] I. Fette and A. Melnikov, “The websocket protocol.” RFC 6455, Internet Engineering Task Force, December 2011.
- [26] I. Fette and A. Melnikov, “The websocket protocol.” RFC 6455, Internet Engineering Task Force, December 2011.
- [27] M. Krakovsky, “Websockets and their impact on reducing bandwidth and latency,” *Journal of Internet Technology*, vol. 22, pp. 221–233, August 2019.
- [28] J. Smith and L. Garcia, “Security in websockets: Challenges and solutions,” *Security Protocols*, vol. 30, no. 2, pp. 155–169, 2020.
- [29] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol http/1.1.” RFC 2616, Internet Engineering Task Force, June 1999.
- [30] Google, “WebRTC - comunicación en tiempo real en la web.” <https://webrtc.org/?hl=es-419>, 2024. Accedido el 2 de octubre de 2024.
- [31] A. Adamatzky, *Physarum machines: computers from slime mould*, vol. 74. World Scientific, 2010.
- [32] A. Adamatzky, “Slime mold solves maze in one pass, assisted by gradient of chemo-attractants,” *IEEE Transactions on NanoBioscience*, vol. 11, pp. 131–134, June 2012.
- [33] J. Jones and A. Adamatzky, “Emergence of self-organized amoeboid movement in a multi-agent approximation of physarum polycephalum,” 2012.
- [34] G. R. Olvera, E. J. S. Méndez, G. J. Martínez, and L. N. O. Moreno, “Modelado del physarum polycephalum con autómatas celulares para el enrutado de robots mensajeros,” Trabajo Terminal 2021 - B013, Instituto Politécnico Nacional, Escuela Superior de Cómputo, México CDMX, enero 2023.
- [35] E. Y. Marín Alavez, “Modelado del physarum polycephalum implementado en robot basado en autómatas celulares,” Mayo 2018. Trabajo Terminal 2017 - A056.

- [36] J. Jones, *From pattern formation to material computation: multi-agent modelling of Physarum Polycephalum*, vol. 15. Springer, 2015.
- [37] Y.-P. Gunji, T. Shirakawa, T. Niizato, and T. Haruna, “Minimal model of a cell connecting amoebic motion and adaptive transport networks,” *Journal of theoretical biology*, vol. 253, pp. 659–67, 05 2008.
- [38] M. Guevara-Bonilla, A. S. Meza-Leandro, E. A. Esquivel-Segura, D. Arias-Aguilar, A. Tapia-Arenas, and F. Masís-Meléndez, “Uso de vehículos aéreos no tripulados (vant’s) para el monitoreo y manejo de los recursos naturales: una síntesis,” *Tecnología en Marcha*, vol. 33, no. 4, pp. 77–88, 2020.
- [39] A. Yehoshua and Y. Edan, “Mobile robots sampling algorithms for monitoring of insects populations in agricultural fields,” 2023.
- [40] R. Bogue, “The role of robots in environmental monitoring,” *Industrial Robot: the international journal of robotics research and application*, vol. 50, no. 3, pp. 369–375, 2023.